System Verilog

Narges Baniasadi

University of Tehran

Spring 2004

System Verilog

- Extensive enhancements to the IEEE 1364 Verilog-2001 standard.
- By Accellera
- More **abstraction**: modeling hardware at the RTL and system level
- **Verification**
- Improved productivity, readability, and reusability of Verilog based code
- Enhanced IP protection

Motivation

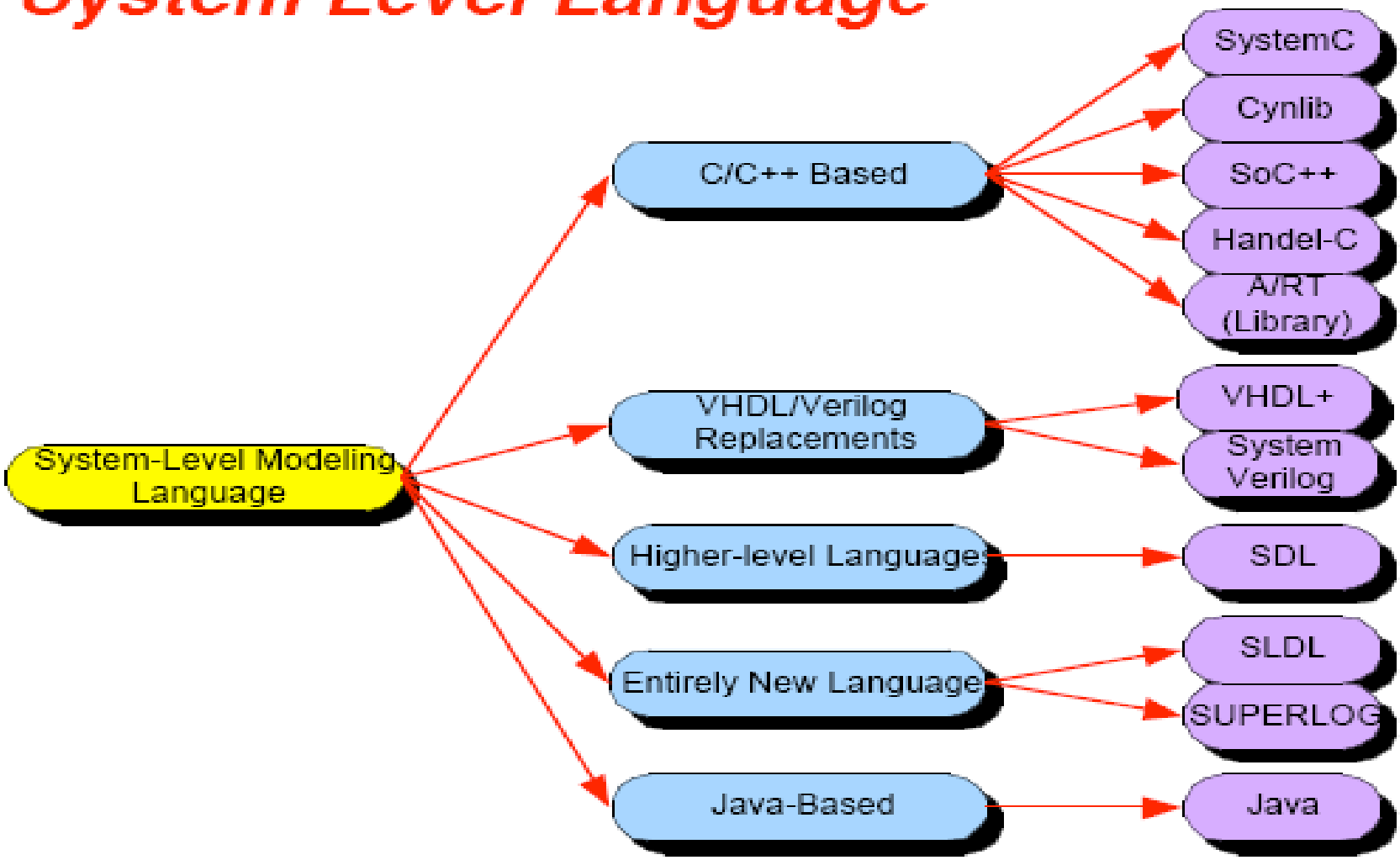
- As design sizes have increased:
  - Design Code size
  - verification code size
  - Simulation time

  Have increased as well.
- Co-design
- Higher Level Design, simulation, synthesis, Test…

Alternatives

- SystemC
  - model full systems at a much higher level of abstraction
- Hardware Verification Languages (HVLs)
  - Verisity's e Synopsys' Vera
  - more concisely describe complex verification routines
  - Increased simulation time
  - Working with multiple languages

System Level Language



Roots

- Accellera chose not to *re-invent the wheel* and *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**

Compatibility with Verilog 2001

- SystemVerilog is fully compatible with the IEEE 1364-2001 Verilog standard
- Warning*:* adds several new keywords to the Verilog language → Identifiers may cause errors: use **Compatibility switches**

Assertions

- Special language constructs to **verify** design behavior; a statement that a specific condition, or sequence of conditions, in a design is true.
- For **documentation** of the assumptions
- Assertions outside of Verilog modules
- Very complex combination of situations

Assertion Types

- Concurrent: the property must be true throughout a simulation

- Procedural:
  - incorporated in procedural code
  - apply only for a limited time

Procedural Assertions

- Immediate : at the time of statement execution
- Strobed : schedule the evaluation of the expression for the end of current timescale to let the glitches settle down
  - Not in functions
- Clocked : being triggered by an event or sampling clock

Immediate assertions

- Severity level
  - **$fatal :** terminates the simulation with an error code. The first argument shall be consistent with the argument to $finish.
  - **$error**
  - **$warning**
  - **$info**

assert (myfunc(a,b)) count1 = count + 1; else ->event1;

[identifier :] assert (expression) [pass_statement] [else fail_statement]

Strobed assertions

- If immediate assertion is triggered by a timing control that happens at the same time as a blocking assignment to the data being tested, there is a risk of the wrong value being sampled.
- ***Pass and Fail statements*** in strobed assertions must not create more events at that time slot or change values.

```
always @(posedge clock)
    a = a + 1; // blocking assignment
always @(posedge clock)
    begin
    ...
    assert (a < b);
    cas:assert_strobe (a < b);

    end
```

System Functions

- **$onehot** (<expression>) returns true if only one and only one bit of expression is high.
- **$onehot0**(<expression>) returns true if at most one bit of expression is low.
- **$inset** (<expression>, <expression> {, <expression> }) returns true if the first expression is equal to at least one of the subsequent expression arguments.
- **$insetz**(<expression>,<expression> {, <expression> }) returns true if the first expression is equal to at least other expression argument.
- **$isunknown**(<expression>) returns true if any bit of the expression is ‘x’.

Assertion example

- a sequence of conditions that span multiple clock cvcles

```
sequence request_check;
   request ##[1:3] grant ##1 !request ##1 !grant;
endsequence

always @(posedge clock)
   if (State == FETCH)
      assert request_check;
```

Interfaces : to encapsulate communication and facilitate “Communication Oriented” design

- Several modules often have many of the same ports
- Abstraction
- Prevent Redundancy in code
- Can include functionality & built-in protocol checking
- the variables and nets in it are assumed to be **inout** ports.

Interface Example 1

```
interface simple_bus; // Define the interface
logic req, gnt;
logic [7:0] addr, data;
logic [1:0] mode;
logic start, rdy;
endinterface: simple_bus

module memMod(simple_bus a, // Use the
    simple_bus interface
input bit clk);
logic avail;
// a.req is the req signal in the 'simple_bus'
    interface
always @(posedge clk) a.gnt <= a.req & avail;
endmodule
```

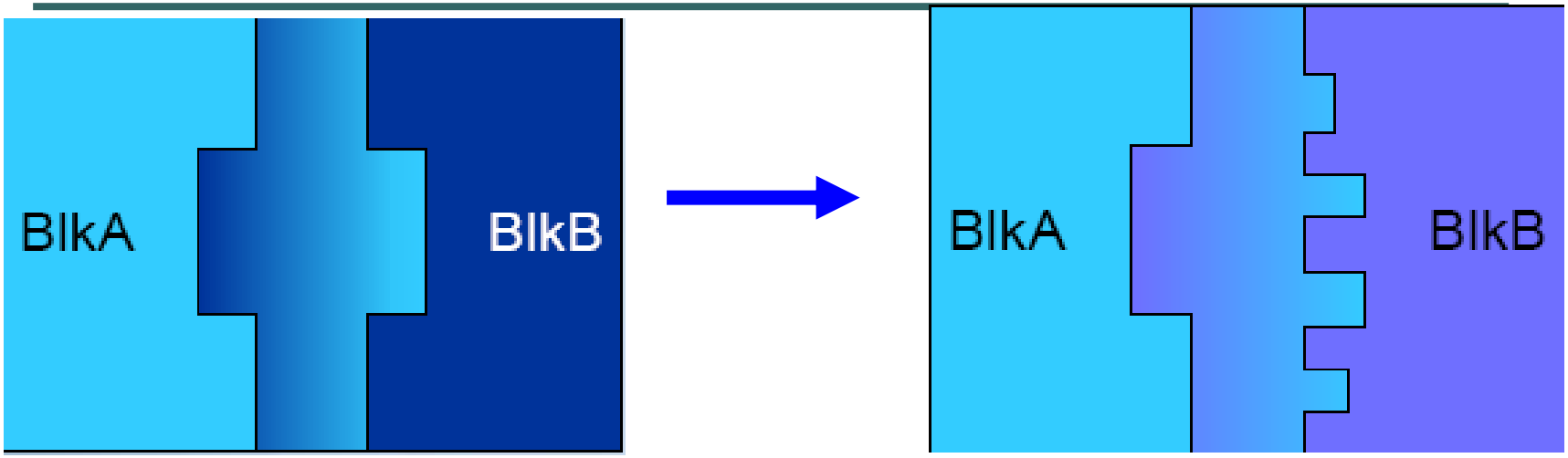
```
module cpuMod(simple_bus b, input
    bit clk);
...
endmodule


module top;
logic clk = 0;
simple_bus sb_intf; // Instantiate the
    interface
memMod mem(sb_intf, clk);
cpuMod cpu(.b(sb_intf), .clk(clk));
endmodule
```

Interface Example 2

```
Interface serial(input bit clk);
logic data_wire;
logic data_start=0;
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)
@(negedgeclk);
for (inti = 0; i <= 31; i++)
begin
d[i] <= data_wire;
@(negedgeclk) ;
end
endtask
endinterface
```

Abstraction & Interfaces



BlkA is unchanged

Data Types

- **More abstract and C like**
  - **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 of any vector width.
  - **logic** — a 4-state unsigned of any vector width, equivalent to the Verilog "reg" data type.
  - **shortreal** — a 2-state single-precision floating-point variable
  - **void** — represents no value**User defined types**
- **User defined types**
  - **Typedef unsigned int uint;**
    uint a, b;

Data Types

- **Enumerated types**
  - **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};
```

Arrays

- *dynamic arrays*
  - one-dimensional arrays where the size of the array can be changed dynamically
- *associative arrays*
  - one-dimensional sparse arrays that can be indexed using values such as enumerated type names.
  - exists(), first(), last(), next(), prev() and delete().

String data type

- contains a variable length array of ASCII characters. Each time a value is assigned to the string, the length of the array is automatically adjusted.
- Operations:
  - standard Verilog operators
  - len(), putc(), getc(), toupper(), tolower(), compare(), icompare(), substr(), atoi(), atohex(), atooct(), atobin(), atoreal(), itoa(), hextoa(), octtoa(), bintoa(), realtoa().

Classes

- data declarations (referred to as *“properties”*),
- tasks and functions for operating on the data (referred to as *“methods”*).
- Classes can have inheritance and public or private protection, as in C++.
- dynamically created, deleted and assigned values. Objects can be accessed via handles, which provide a safe form of pointers.
- Memory allocation, de-allocation and garbage collection are automatically handled.
- dynamic nature: ideal for testbench modeling. Not Synthesizable
- verification routines and highly abstract system-level modeling.

Class example:

```
class Packet;
   bit [3:0] command;
   bit [39:0] address;
   bit [4:0] master_id;
   integer time_requested;
   integer time_issued;
   integer status;

   function new();
      command = 4'hA;
      address = 40'hFE;
      master_id = 5'b0;
   endfunction

   task clean();
      command = 4'h0; address = 40'h0;
      master_id = 5'b0;
   endtask
endclass
```

Relaxed data type rules

- Net data types (such as wire, wand, wor)
- Variables (such as reg, integer)

Allowing variable types to be used in almost any context.

Type Casting

```
int'(2.0 * 3.0) //cast result to int
mytype'(foo)    //cast foo to the user-defined type of mytype
17'( x - 2)     //cast the operation to 17 bits
signed'(x)      //cast x to a signed value
```

Global declarations

- Any declarations outside of a module boundary are in the global, or **root**, name space.
- Variables, type definitions, functions

New Operators

- ++ and -- increment and decrement operators
- +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, <<<= and >>>= assignment operators

Time units

- Not a compiler directive
  - module and the files can be compiled in any order
- Time unit/precision can be declared as part of the module.

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

Unique and priority decision statements

- if-else and case statements can be a source of mismatches between RTL simulation & synthesis
- The synthesis full_case and parallel_case pragmas can lead to further mismatches if improperly used, as they affect synthesis but not simulation.
- SystemVerilog adds the ability to explicitly specify when each branch of a decision statement is unique or requires priority evaluation, using the keywords "unique" and "priority."

```
priority casez(a)
   3'b00?: y = in1; // a is 0 or 1
   3'b0??: y = in2; //a is 2 or 3;
   default: y = in3; //a is any other value
endcase
```

Enhanced for loops

- Allow the loop control variable to be declared as part of the for loop, and allows the loop to contain multiple initial and step assignments.
  - **for (int** i=1, **shortint** count=0; i*count < 125; i++, count+=3)
- **Bottom testing loops.**
  - adds a do-while loop, which tests the loop condition at the end of executing code in the loop.
- **Jump statements.**
  - adds the C "break" and "continue" keywords, which do not require the use of block names, and a "return" keyword, which can be used to exit a task or function at any point.
- **Final blocks.**
  - execute at the very end of simulation,
  - can be used in verification to print simulation results, such as code coverage reports

Hardware-specific procedures

- Instead of inferring the intent of the always procedure from the sensitivity list and the statements within the procedure explicitly indicate the intent of the logic:
  - always_ff — the procedure is intended to represent sequential logic
  - always_comb —: the procedure is intended to represent combinational logic
  - always_latch — the procedure is intended to represent latched logic. For example:

Task and function enhancements

- Function return values can have a "void" type. Void functions can be called the same as a Verilog task.
- Functions can have any number of inputs, outputs and inouts, including none.
- Values can be passed to a task or function in any order, using the task/function argument names.
- Task and function input arguments can be assigned a default value as part of the task/function declaration. This allows the task or function to be called without passing a value to each argument.
- Task or function arguments can be passed by reference, instead of copying the values in or out of the task or function. To use pass by reference, the argument direction is declared as a "ref," instead of input, output or inout.

Inter-process synchronization

- **semaphore :**
  - as a bucket with a fixed number of “keys.”
  - built-in methods : new(), get(), put() and try_get().
- **mailbox**
  - allows messages to be exchanged between processes. A message can be added to the mailbox at anytime by one process, and retrieved anytime later by another process.
  - Mailboxes behave like FIFOs (First-In, First-Out).
  - built-in methods: new(), put(), tryput(), get(), peek(), try_get() and try_peek().
- **Event**
  - The Verilog "event" type is a momentary flag that has no logic value and no duration. If a process is not watching when the event is triggered, the event will not be detected.
  - SystemVerilog enhances the event data type by allowing events to have persistence throughout the current simulation time step. This allows the event to be checked after it is triggered.

Constrained random values

- Verilog $random : little control
- rand and randc: classes with methods to set seed values and to specify various constraints on the random values that are generated

Testbench program block

- Declared between the keywords "program" and "endprogram."
- Contains a single initial block.
- Executes events in a “reactive phase” of the current simulation time, appropriately synchronized to hardware simulation events.
- Can use a special "$exit" system task that will wait to exit simulation until after all concurrent program blocks have completed execution (unlike "$finish," which exits simulation immediately).

Clocking domains

- Clocking domains allow the testbench to be defined using a cycle-based methodology, rather than the traditional event-based methodology
- A clocking domain can define detailed skew information
- avoiding race conditions with the design

Clocking Domain

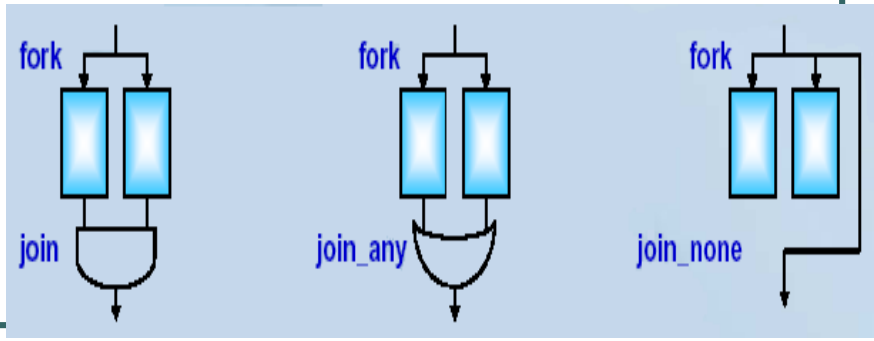
```
clocking bus @(posedge clk);
   default input #2ns output #1ns;    //default I/O skew
   input      enable, full;
   inout      data;
   output        empty;
   output        #6ns reset;          //reset skew is different than default
endclocking

initial //Race conditions with Verilog; no races with SystemVerilog
   begin
      @(posedge clk)  input_vector = ...       //drive stimulus onto design
      @(posedge clk)  $display(chip_output);   //sample result
                      input_vector = ...       //drive stimulus onto design
      @(posedge clk)  $display(chip_output);   //sample result
                      input_vector = ...       //drive stimulus onto design
   end
```

Direct Programming Interface (DPI)

- To directly call functions written C, C++ or SystemC, without having to use the complex Verilog Programming Language Interface (PLI).
- Values can be passed and received directly to/from the foreign language function.
- Gives the foreign language functions access to simulation events and simulation time.
- A bridge between high-level system design using C, C++ or SystemC and lower-level RTL and gate-level hardware design.

Enhanced fork-join

- **join_none** — statements that follow the fork-join_none are not blocked from execution while the parallel threads are executing. Each parallel thread is an independent, dynamic process.
- **join_any** — statements which follow a fork-join_any are blocked from execution until the first of any of the threads has completed execution



Processes

- Static processes
  - always
  - initial
  - fork
- Dynamic processes
  - introduced by process.
- SystemVerilog creates a thread of execution for each initial or always block, for each parallel statement in a fork...join block and for each dynamic process

Dynamic Processes

- A fork without a join
- Does not block the flow of execution of statements within the procedure or task: A separate thread
- Multi-threaded processes

```
task monitorMem(input int address);
process forever @strobe $display("address %h data %h", mem[address]
    );
endtask
```

Enhanced IP Protection

- Modules can be declared within modules and these nested modules are local to their parent modules.
- Other parts of the design can not see local modules.

References:

- An overview of SystemVerilog 3.1
  By Stuart Sutherland, EEdesign
  May 21, 2003 (4:09 PM)

- System Verilog 3.0 LRM