EEL 4783: HDL in Digital System Design

Lecture x: Introduction to SystemVerilog*

Prof. Mingjie Lin

* largely adopted from University of Washington EE400/590 Dean Wen Chen

Outline

- Introduction
- SystemVerilog enhancements overview
- Conclusion

Introduction

- What is Verilog HDL?
  - Verilog HDL is a Hardware Description Language that can be used to model a digital system at various levels of abstraction.
- What is SystemVerilog?
  - SystemVerilog is an extensive set of enhancements to the IEEE 1364 Verilog-2001 standard.

A problem that needed solving

- As design sizes have increased, several things increased as well:
  - Number of lines of RTL design code size
  - Verification code size
  - Simulation time

Alternatives to the problem

- SystemC:
  - modeling full systems at a much higher level of abstraction.
- Hardware Verification Languages (HVLs)
  - Such as Verisity's e and Synopsys' Vera
  - More concisely describe complex verification routines
  - Require to work with multiple languages
  - Increased simulation time.

SystemVerilog's roots

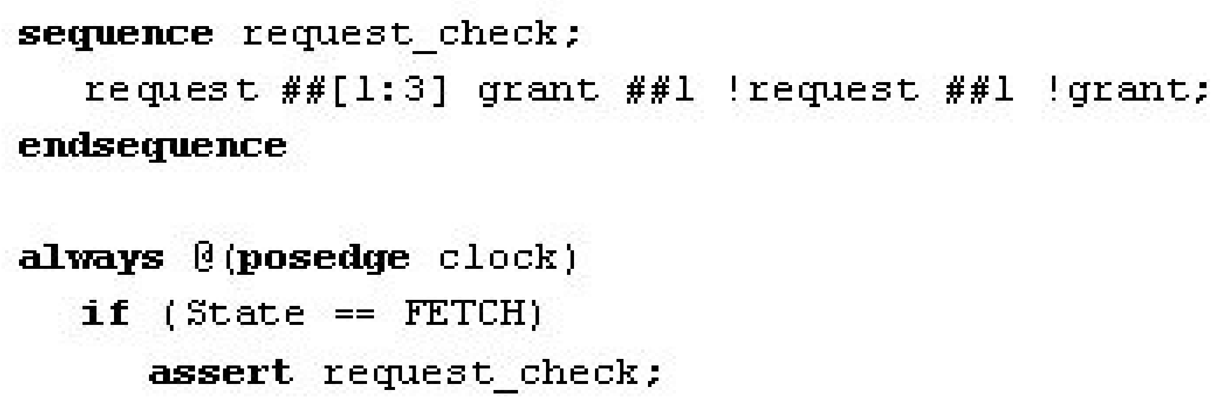
- Instead of re-invent the wheel, Accellera relied on donations of technology from a number of companies.
  - **High-level modeling constructs** : Superlog language developed by Co-Design
  - **Testbench constructs** : Open Vera language and VCS DirectC interface technology by Synopsys
  - **Assertions**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

Compatibility with Verilog-2001

- fully compatible with the IEEE 1364-2001 Verilog standard.
- There is one caveat to this backward compatibility. SystemVerilog adds several new keywords to the Verilog language. Identifier maybe used in an existing model. Compatibility switches can be used to deal with this problem.

Assertions

- Special language constructs to verify design behavior.
- Example:



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

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

Assertions (Cont..)

- Assertions can be defined outside of Verilog modules, and then bind them to a specific module or module instance. This allows verification engineers to add assertions to existing Verilog models, without having to change the model in any way

Interfaces

- High level of abstraction for module connections.
- Modules can use an interface the same as if it were a single port.
- Can be considered a bundle of wires. Interfaces go far beyond just representing bundles of interconnecting signals, however. An interface
- Can also include functionality and built-in protocol checking.

Interface Example

```
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
```

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)

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;

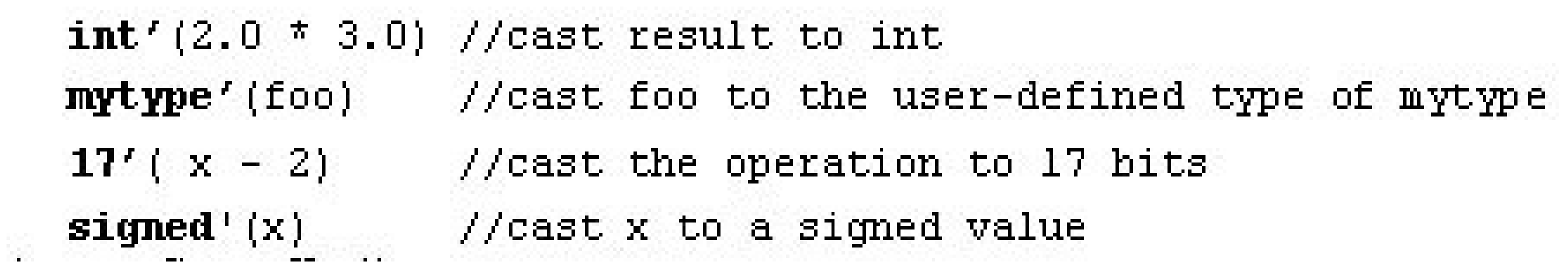
Data Types (Cont..)

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

Casting

- SystemVerilog adds the ability to change the type, vector size or "signedness" of a value using a cast operation. To remain backward compatible with the existing Verilog language, casting in SystemVerilog uses a different syntax than C.



```
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
```

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().

Classes

- Can contain data declarations (referred to as "properties")
- Can contain functions for operating on the data (referred to as "methods").
- Can have inheritance and public or private protection, as in C++.
- Allow objects to be 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, preventing the possibility of memory leaks.
- Classes are dynamic by nature, instead of static. They are ideal for test-bench modeling. Therefore, they are not considered synthesizable constructs.
- Intended for 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
```

String data type

- Defined as a built-in class.
- The 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.

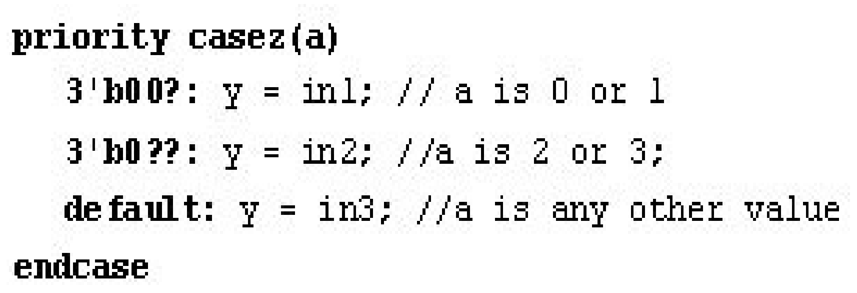
- Operations:
  - Standard Verilog operators: =, ==, !=, <, <=, >, >=, {,}, {{}}.
  - Methods: len(), putc(), getc(), toupper(), tolower(), compare(), icompare(), substr(), atoi(), atohex(), atooct(), atobin(), atoreal(), itoa(), hextoa(), octtoa(), bintoa() and realtoa().

Operators

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

Unique and priority decision statements

- 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." These keywords affect simulators, synthesis compilers, formal verifiers and other tools, ensuring that all tools interpret the model the same way.



```
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 "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, just before simulation exits. Final blocks can be used in verification to print simulation results, such as code coverage reports.

Hardware-specific procedures

- adds three new procedures to 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.

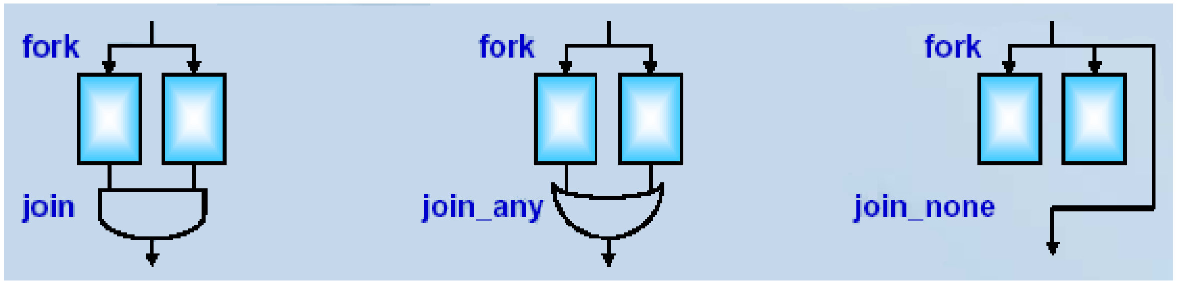
```
always_comb
  if (sel) y = a;
  else y = b;
```

Task and function enhancements

- Function return values can have a "void" return 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. The syntax is the same as named module port connections.
- 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.

Enhanced fork-join

- Adds fork-join_none, and fork-join_any blocks.
  - **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.

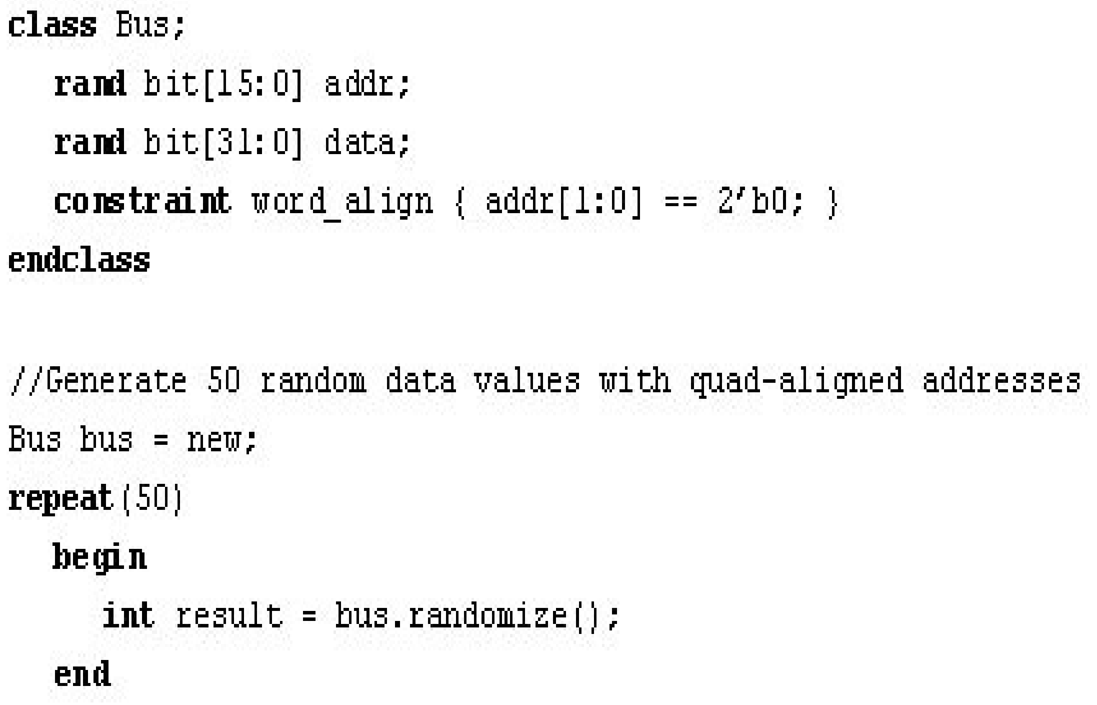


Inter-process synchronization

- **semaphore :**
  - Serve as a bucket with a fixed number of "keys."
  - Built-in methods for working with semaphores: 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.
  - 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

- adds two random number classes, "rand" and "randc,"



```
class Bus;
   rand bit[15:0] addr;
   rand bit[31:0] data;
   constraint word_align { addr[1:0] == 2’b0; }
endclass

//Generate 50 random data values with quad-aligned addresses
Bus bus = new;
repeat(50)
   begin
      int result = bus.randomize();
   end
```

Testbench program block

- 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 of defining specific transition times for each test signal.
- A clocking domain can define detailed skew information
- Greatly simplify defining a testbench that does not have race conditions with the design being tested.

Clocking domains (Cont..)

```
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 directly to the foreign language function, and values can be received from the function.
- The foreign language function can also call Verilog tasks and functions, which gives the foreign language functions access to simulation events and simulation time.
- The SystemVerilog DPI provides a bridge between high-level system design using C, C++ or SystemC and lower-level RTL and gate-level hardware design.

Conclusion

- SystemVerilog provides a major set of extensions to the Verilog-2001 standard. These extensions allow modeling and verifying very large designs more easily and with less coding.
- SystemVerilog extends the modeling aspects of Verilog, and adds a Direct Programming Interface which allows C, C++, SystemC and Verilog code to work together without the overhead of the Verilog PLI.
- SystemVerilog bridges the gap between hardware design engineers and system design engineers. SystemVerilog also significantly extends the verification aspects of Verilog by incorporating the capabilities of Vera and powerful assertion constructs.
- Adding these SystemVerilog extensions to Verilog creates a whole new type of engineering language, an HDVL, or Hardware Description and Verification Language. This unified language will greatly increase the ability to model huge designs, and verify that these designs are functionally correct.

References

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

- System Verilog by Narges Baniasadi, University of Tehran Spring 2004