An Overview of SystemVerilog for Design and Verification

Vighnesh Iyer, EECS 251B
Intention of this Lecture

- We use Chisel for all RTL written at Berkeley
  - Why bother with SystemVerilog?
- SystemVerilog is the de-facto industry standard
  - SV/UVM is used for (nearly) all industry verification
  - You will be asked about it in interviews
- Understand basic dynamic verification concepts
- Understand existing SystemVerilog code
- Inspire extensions to HDLs

SystemVerilog (SV) is an IEEE Standard 1800
https://standards.ieee.org/project/1800.html

Universal Verification Methodology (UVM) is a standard maintained by Accellera
https://www.accellera.org/downloads/standards/uvm
What is SystemVerilog

● IEEE 1800 standard
● A massive extension of Verilog with new constructs for design and verification
  ○ New data types (for RTL and testbenches)
  ○ OOP support
  ○ Constrained random API
  ○ Specification language (SystemVerilog Assertions (SVA))
  ○ Coverage specification API
● Fixing warts in Verilog
  ○ Synthesis - simulation mismatch
  ○ Verilog was initially developed as a simulation language; synthesis emerged later
SystemVerilog for Design
### Ending the Wire vs. Reg Confusion

#### Verilog-2005
- **wire** for LHS of assign statements
- **reg** for LHS of code inside `always @` blocks

```verilog
wire a;
reg b, c;
assign a = ____;
always @(*) b = ____;
always @(posedge clk) c <= ____;
```

#### SystemVerilog
- **logic** for LHS of assign statements
- **logic** for LHS of code inside `always @` blocks

```verilog
logic a, b, c;
assign a = ____;
always @(*) b = ____;
always @(posedge clk) c <= ____;
```

Both: the **containing statement** determines if the net is the output of a **register** or **combinational** logic.
Signal Your Intent With Specific Always Blocks

**Verilog-2005**

```
always @(*) begin
  if (x) b = a;
  else  b = !a;
end

always @posedge clk begin
  if (x) c <= !a;
  else  c <= a;
end
```

**SystemVerilog**

```
always_comb begin
  if (x) b = a;
  else  b = !a;
end

always_ff @posedge clk begin
  if (x) c <= !a;
  else  c <= a;
end
```

Coding style is used to verify that c infers as a register and b as comb logic.

New *always_comb* and *always_ff* statements for safety.
Autoconnect (Implicit Port Connections)

- How many times have you done this?

```verilog
module mod (input a, b, output c); endmodule

reg a, b; wire c;
mod x (.a(a), .b(b), .c(c));
```

- If the net names and their corresponding port names match, there’s a shortcut

```verilog
mod x (.a, .b, .c);
```

- In SystemVerilog, there’s a more concise shortcut

```verilog
mod x (.*);
```

- Implicit connections only work if port names and widths match
Use Enums Over localparams

**Verilog-2005**

```verilog
localparam STATE_IDLE = 2'b00;
localparam STATE_A = 2'b01;
localparam STATE_B = 2'b10;
reg [1:0] state;

always @(posedge clk) begin
  case (state)
    STATE_IDLE: state <= STATE_A;
    STATE_A: state <= STATE_B;
    STATE_B: state <= STATE_IDLE;
  endcase
end
```

**SystemVerilog**

```verilog
typedef enum logic[1:0] {
  STATE_IDLE, STATE_A, STATE_B
} state_t;
state_t state;

always_ff @(posedge clk) begin
  case (state)
    STATE_IDLE: state <= STATE_A;
    STATE_A: state <= STATE_B;
    STATE_B: state <= STATE_IDLE;
  endcase
end
```

Enums automatically check whether all values can fit. Can be used as a net type. Adds **semantic meaning** to constants.
More on Enums

- Common to use enums for attaching semantic strings to values

```vhdl
typedef enum logic {
    READ, WRITE
} mem_op;

module memory (  
    input [4:0] addr,
    input mem_op op,
    input [31:0] din,
    output logic [31:0] dout
);
```

- Note that input/output net types are by default ‘wire’, you can override them as logic
Even More on Enums

- You can force enum values to be associated with a specific value
  - To help match up literals for a port that doesn’t use enums
    ```c++
    typedef enum logic [1:0] { STATE_IDLE=3, STATE_A=2, STATE_B=1 } state_t
    ```

- You can generate N enum values without typing them out
  ```c++
  typedef enum logic [1:0] { STATE[3] } state_t
  // STATE0 = 0, STATE1 = 1, STATE2 = 2
  ```

- You can generate N enum values in a particular range
  ```c++
  typedef enum logic [1:0] { STATE[3:5] } state_t
  // STATE3 = 0, STATE4 = 1, STATE5 = 2
  ```
Even More on Enums

- Enums are a first-class datatype in SystemVerilog
  - Enum instances have native functions defined on them
    - `next()`: next value from current value
    - `prev()`: previous value from current value
    - `num()`: number of elements in enum
    - `name()`: returns a string with the enum’s name (useful for printing using `$display`)

- They show up in waveforms
  - No more confusion trying to correlate literals to a semantic name

- They are weakly typechecked
  - You can’t assign a binary literal to a enum type net (without a warning)
Multidimensional Packed Arrays

- **Packed** dimensions are to the left of the variable name
  - Packed dimensions are contiguous (e.g. `logic [7:0] a`)
- **Unpacked** dimensions are to the right of the variable name
  - Unpacked dimensions are non-contiguous (e.g. `logic a [8]`)

```verbatim
logic [31:0] memory [32];
// memory[0] is 32 bits wide
// cannot represent more than 1 dimension in memory[0]
// can’t easily byte address the memory

logic [3:0][7:0] memory [32];
// memory[0] is 32 bits wide
// memory[0][0] is 8 bits wide
// memory[0][1] is 8 bits wide
```
Structs

- Similar to Bundle in Chisel
  - Allows designer to group nets together, helps encapsulation of signals, easy declaration
  - Can be used within a module or in a module’s ports
  - Structs themselves can’t be parameterized
    - but can be created inside a parameterized module/interface

```chisel
typedef struct packed {
  logic [31:0] din,
  logic [7:0] addr,
  logic [3:0] wen,
  mem_op op
} ram_cmd;

module ram (ram_cmd cmd,
    logic [31:0] dout
);
```

```chisel
typedef struct packed {
  logic [31:0] din,
  logic [7:0] addr,
  logic [3:0] wen,
  mem_op op
} ram_cmd;

always_ff @(posedge clk) begin
  a.din  <= ____;
  a.addr <= ____;
  a.wen  <= ____;
  a.op   <= ____;
end
```
Interfaces

- Interfaces allow designers to group together ports
  - Can be parameterized
  - Can contain structs, initial blocks with assertions, and other verification collateral
  - **Simplify connections** between parent and child modules

```verilog
interface ram_if #(int addr_bits, data_bits) (input clk);
  logic [addr_bits-1:0] addr;
  logic [data_bits-1:0] din;
  logic [data_bits-1:0] dout;
  mem_op op;
endinterface

module ram (#(int addr_bits, data_bits) intf);
  ram_if intf;

  always_ff @(posedge intf.clk)
    intf.dout <= ram[intf.addr];
endmodule

module top();
  ram_if #(8, 32) intf();
  ram r (.intf(intf));
  assign intf.din = ____;
endmodule
```
Modports

- But I didn’t specify the direction (input/output) of the interface ports!
  - This can cause multi-driver issues with improper connections
- Solution: use `modports`

```verilog
text
interface ram_if #(int addr_bits, data_bits)
(input clk);
    modport target (
        input addr, din, op, clk,
        output dout
    );

    modport controller (
        output addr, din, op,
        input dout, clk
    );
endinterface

module ram (
    ram_if.target intf
);
    always_ff @(posedge intf.clk)
        intf.dout <= ram[intf.addr];
endmodule
```
Typedefs (Type Aliases)

- You probably saw ‘typedef’ everywhere
  - typedefs make it easier to reuse user-defined types
  - They are type aliases
- Just like with enums, they can also help to attach semantic meaning to your design

```c
typedef signed logic [7:0] sgn_byte;
typedef unsigned logic [3:0] cache_idx;
```
Sometimes we want to make sure synthesis infers parallel logic vs priority mux

The ‘unique’ keyword applied to a ‘if’ or ‘case’ statement

○ Adds simulation assertions to make sure only one branch condition is true
○ Tells synthesis tools to operate under that assumption
○ Legacy: ‘synopsys parallel_case full_case’

```vhdl
always_comb begin
    unique if (x == 2'b10) a = ____;
    else if (y && x == 2'b11) a = ____;
    else a = ____;
end
```
Packages / Namespacing

- Verilog has a global namespace
  - Often naming conflicts in large projects
  - `include is hacky and requires `ifdef guards
- SystemVerilog allows you to encapsulate constructs in a package
  - modules, functions, structs, typedefs, classes

```verilog
package my_pkg;
  typedef enum logic [1:0] { STATE[4] } state_t;
  function show_vals();
    state_t s = STATE0;
    for (int i = 0; i < s.num; i = i + 1) begin
      $display(s.name());
      s = s.next();
    end
  endfunction
endpackage

import my_pkg::*;

module ex (input clk);
  state_t s;
  always_ff @(posedge clk) begin
    s <= STATE0;
  end
endmodule
```
SystemVerilog for Verification
Overview

- The SystemVerilog spec for verification is massive
  - We can’t cover everything in one lecture
- New data structures for writing testbenches
- OOP
- SystemVerilog Assertions
- Coverage API
- Constrained random
New Data Types

- **bit, shortint, int, longint**
  - 2-state types (1/0)
  - Contrast to ‘logic’ which is a 4-state type (1/0/x/z)
- **string**
  - Now natively supported, some helper methods are defined on string (e.g. substr)
Dynamic Arrays

● Typical Verilog arrays are fixed length at compile time

```verilog
bit [3:0] arr [3]; // a 3 element array of 4 bit values
arr = {12, 10, 3}; // a literal array assignment
```

● Dynamic arrays are sized at runtime
  ○ Useful for generating variable length stimulus

```verilog
bit [3:0] arr []; // a dynamic array of 4 bit values
initial begin
  arr = new[2]; // size the array for 2 elements
  arr = {12, 10}; // literal assignment
  arr = new[10];
  arr[3] = 4;
end
```
Queues

- Similar to lists in Scala and Python
  - Useful for hardware modeling (FIFO, stack) - process transactions sequentially

```verilog
bit [3:0] data[:] = '{1, 2, 3, 4}; // initialized queue
```

```python
data[0] // first element
data[:] // last element
data.insert(1) // append element
data[1:] // queue slice excluding first element
x = data.pop_front() // pops first element of queue and returns it
data = {} // clear the queue
```
Associative Arrays

- Similar to Python dicts or Scala Maps
  - Can be used to model a CAM or look up testbench component settings

```java
int fruits [string];
fruits = {'apple': 4, 'orange': 10};

fruits['apple']
fruits.exists('lemon')
fruits.delete('orange')
```
Clocking Blocks

- There is often confusion when you should drive DUT inputs and sample DUT outputs relative to the clock edge
  - Solution: encode the correct behavior in the interface by using clocking blocks

```
interface ram_if #(int addr_bits, data_bits) (input clk);
  logic [addr_bits-1:0] addr;
  logic [data_bits-1:0] din;
  logic [data_bits-1:0] dout;
  mem_op op;

clocking ckb @(posedge clk)
  default input #1step output negedge;
  input dout;
  output addr, din, op;
endclocking
endinterface
```

- Input/output is from the perspective of the testbench
- Can use any delay value or edge event as skew
- intf.ckb.din = 32'd100; @(intf.ckb); x = intf.ckb.dout;
OOP in SystemVerilog

- SystemVerilog has your typical object-oriented programming (OOP) constructs
  - Classes, constructors, type generics, inheritance, virtual methods/classes, polymorphism

```verilog
class Message;
  bit [31:0] addr;
  bit [3:0] wr_strobe;
  bit [3:0] burst_mode;
  bit [31:0] data [4];

  function new (bit [31:0] addr, bit [3:0] wr_strobe = 4'd0);
    this.addr = addr;
    this.wr_mode = wr_mode;
    this.burst_mode = 4'b1010;
    this.data = '{0, 0, 0, 0};
  endfunction
endclass

initial begin
  msg = new Message(32'd4, 4'b1111);
  $display(msg.burst_mode);
end
```
More OOP

● You can extend a class as usual
  ○ class ALUMessage extends Message
  ○ call .super() to access superclass functions
  ○ Polymorphic dynamic dispatch works as usual

● You can declare classes and functions ‘virtual’
  ○ Forces subclasses to provide an implementation
  ○ Prevents instantiation of abstract parent class

● Class members can be declared ‘static’
  ○ The member is shared among all class instances

● OOP constructs are used to:
  ○ Model transactions
  ○ Model hardware components (hierarchically and compositionally)
Type Generic Classes

- Classes can have parameters, just like modules
  - They can be ints, strings, or **types**
    - Can't define type bounds on type parameters
  - Class parameters can have default values
  - Classes can have constructors (the 'new' function) just like in Java

```
class FIFO #(type T = int, int entries = 8);
  T items [entries];
  int ptr;

  function void push(T entry);
  function T pull();
endclass
```

```FIFO #(T = logic [7:0], entries = 128) f = new(); // holds 128 8-bit numbers
FIFO #(T = mem_cmd) f = new(); // holds 8 mem_cmd structs```
SystemVerilog Assertions (SVA)
SystemVerilog Assertions (SVA)

- The most complex component of SystemVerilog
  - Entire books written on just this topic
- **SVA**: a temporal property specification language
  - Allows you to formally specify the expected behavior of signals in your design
- You are already familiar with ‘assert’ (so-called ‘immediate assertions’)

```verilog
module testbench();
  dut d (.addr, .dout);

initial begin
  addr = $h40;
  assert (dout == $hDEADBEEF);
end
endmodule
```

- But how do I express properties that involve time-domain behaviors?
- Can I express these properties (e.g. req-ack) in a concise way?
Concurrent Assertions

- Concurrent assertions are constantly monitored by the RTL simulator
  - Often embedded in the DUT RTL or an interface

```verilog
cpu();
  assert property @(posedge clk) (mem_addr[1:0] != 2'd0) && load_word |-> unaligned_load
  assert property @(posedge clk) opcode == 0 |-> take_exception
  assert property @(posedge clk) mem_stall |=> $stable(pc)
endmodule
```

- Properties are evaluated on an event (e.g. clock edge)
- | ->: same-cycle implication
- | =>: next-cycle implication
- These properties can also be formally verified
You can call a system function in an SVA expression to simplify checking historical properties:

- $stable(x)$: indicates if $x$ was unchanged from the previous clock cycle
- $rose(x)$
- $fell(x)$
- $past(x)$: gives you the value of $x$ from 1 cycle ago
  - $rs1_{mem} == $past(rs1_{ex})$
Sequences

- Properties are made up of sequences + an implication

module cpu();
  sequence stall
    mem_stall;
  endsequence

  sequence unchanged_pc
    ##1 $stable(pc);
  endsequence

  property stall_holds_pc
    @(posedge clk) stall |-> unchanged_pc;
  endproperty

  assert property (stall_holds_pc);
endmodule

- Many interfaces come with sequence libraries to build complex properties
Sequence Combinators

- Sequences are the core of SVA: they describe temporal RTL behavior
- Sequences can be combined with temporal operators

\[ a \, \#\#1 \, b \] // a then b on the next cycle
\[ a \, \#\#N \, b \] // a then b on the Nth cycle
\[ a \, \#\#[1:4] \, b \] // a then b on the 1-4th subsequent cycle
\[ a \, \#\#[2:5] \, b \] // a then b after 2 or more cycles

\[ s1 \, \text{and} \, s2 \] // sequence s1 and s2 succeed
\[ s1 \, \text{intersect} \, s2 \] // sequence s1 and s2 succeed and end at the same time
\[ s1 \, \text{or} \, s2 \] // sequence s1 or s2 succeeds

- Sequences are combined with an implication to form a property
  - There's a lot more to SVA
Coverage APIs
Coverage

- You’re probably familiar with software coverage tools
  - Track if a line of source code is hit by the unit tests
- Coverage is used to measure the thoroughness of the test suite
  - Are all the interesting cases in the code exercised?
- RTL coverage comes in two forms
  - Structural coverage: line, toggle, condition
  - Functional coverage: did a particular uArch feature specified by the DV engineer get exercised?
    - e.g. cache eviction, misaligned memory access, interrupt, all opcodes executed
Property Coverage

● Any SVA property can be tracked for coverage
  ○ Instead of ‘assert property’ use ‘cover property’

```verilog
define property req_ack;
    req ##[1:10] ack
endproperty
cover property (req_ack)
```

● Property covers are used in RTL to check that some multi-cycle uArch behavior is exercised
  ○ e.g. did this req-ack handshake ever occur?
  ○ e.g. did a branch mispredict and predictor update happen?
Coverpoints and Covergroups

- Coverpoints track coverage of a single net
  - e.g. FSM state, control signals, data buses
- Covergroups group together coverpoints with a sampling event
  - Can be used in RTL and in testbench code

```verilog
module cpu ();
    logic [5:0] rs1, rs2;
    logic [2:0] funct3;

    covergroup c @(posedge clk);
        coverpoint rs1;
        coverpoint funct3;
    endgroup

endmodule
```
Coverpoint Bins

- Sometimes we don’t want to track each value a net can take on individually
  - Use the bins API to group some values together

```vhdl
module alu(input [31:0] a, input [31:0] b, input [3:0] op, output [31:0] out);
  covergroup c();
  coverpoint a {
    bins zero = {0};
    bins max = {32'hffff_ffff};
    // automatically allocate 100 uniformly sized bins for the remaining numbers
    bins in_the_middle[100] = {{1:32'hffff_ffff - 1}};
  }
endgroup
endmodule
```
Transaction-Level Modeling
Transactions

- Our testbenches are usually written at cycle-granularity
  - Leads to mixing of driving/monitoring protocols, timing details, golden models, and stimulus
  - Each of these concerns should be separated
- Model a single interaction with the DUT as a ‘transaction’
  - It can take multiple cycles
- We can build a stimulus generator and golden model at transaction-level

```vhdl
class MemReqTx();
    bit [31:0] addr;
    bit [31:0] wr_data;
    mem_op op;
endclass

class MemRespTx();
    bit [31:0] rd_data;
endclass

class Mem();
    bit [31:0] ram [];
    function MemRespTx processTx(MemReqTx tx);
endclass
```
VIPs and Testbench Architecture

- Verification IPs consist of tasks that encode
  - How to drive transactions into an interface at cycle granularity
  - How to translate cycle granularity interface activity into transactions

- A testbench
  - Generates stimulus
  - Generates golden DUT behavior
  - Simulates actual DUT behavior
  - Checks correctness
Random Transaction Generation

- How do we generate transaction-level stimulus?
- SystemVerilog class members can be prefixed with the `rand` keyword
  - These fields are marked as randomizable

```systemverilog
class MemReqTx();
  rand bit [31:0] addr;
  rand bit [31:0] wr_data;
  rand mem_op op;
endclass

initial begin
  MemReqTx tx = new();
  tx.randomize();
end
```
Constrained Random

- You can constrain the random fields of a class inside or outside the class
  - You can add ad-hoc constraints when calling `.randomize`

```verbatim
class cls;
  rand bit [7:0] min, typ, max;

  constraint range {
    0 < min; typ < max; typ > min; max < 128;
  }
  extern constraint extra;
endclass

constraint cls::extra { min > 5; }
initial begin
  cls = new();
  cls.randomize() with { min == 10; }
end
```
Randomization of Variable Length Data Structures

```plaintext
class Packet;
    rand bit [3:0] data [];

    constraint size { data.size() > 5; data.size() < 10; }

    constraint values {
        foreach(data[i]) {
            data[i] == i + 1;
            data[i] inside {[0:8]};

        }
    }
endclass
```

- Many things I haven’t discussed
  - Biasing and distributions, soft constraints, disables, solve before, implications, dynamic constraint on/off
Mailboxes for Inter-Thread Communication

- Mailboxes are like golang channels
  - Bounded queues that allow one simulation thread to send data to another

```verilog
module example;
    mailbox #(int) m = new(100);

    initial begin
        for (int i = 0; i < 200; i++)
            #1 m.put(i);
    end

    initial begin
        for (int i = 0; i < 200; i++) begin
            int i; #m.get(i);
            $display(i, m.num());
            end
    end
endmodule
```
Testbench Example
Register Bank

- Let’s test a simple register bank
  - Works like a memory
  - Multi-cycle (potentially variable) read/write latency
  - Uses a ready signal to indicate when a new operation (read/write) can begin

```vhdl
interface reg_if (input clk);
  logic rst;
  logic [7:0] addr;
  logic [15:0] wdata;
  logic [15:0] rdata;
  mem_op op;
  logic en;
  logic ready;
endinterface

module regbank (reg_if.target if);
  // implementation
  endmodule

// Regbank transaction
class regbank_tx;
  rand bit [7:0] addr;
  rand bit [15:0] wdata;
  bit [15:0] rdata;
  rand bit wr;
endclass
```
class driver;
    virtual reg_if vif;
    mailbox drv_mbx;

    task run();
        @(vif.drv_cb);
        forever begin
            regbank_tx tx;
            drv_mbx.get(tx);
            vif.drv_cb.en <= 1;
            vif.drv_cb.addr <= tx.addr;
            // assign op and wdata
            @(vif.drv_cb);
            while (!vif.drv_cb.ready)
                @(vif.drv_cb)
        end
    endtask
endclass

class monitor;
    virtual reg_if vif;
    mailbox mon_mbx;

    task run();
        @(vif.mon_cb);
        forever begin
            if (vif.en) begin
                regbank_tx tx = new();
                tx.addr = vif.mon_cb.addr;
                // assign op and wdata
                @(vif.mon_cb);
                if (vif.mon_cb.op == READ) begin
                    tx.rdata = vif.mon_cb.rdata;
                end
            end
            mon_mbx.put(tx);
        end
    endtask
endclass
Top-Level

- A rough sketch of the testbench top

```verilog
module tb();
    regbank dut (.*);

initial begin
    // initialize driver/monitor classes
    regbank_tx stim [100];
    stim.randomize();
    fork
        drv.run();  mon.run();
    join_none
        drv.drv_mbx.put(stim);
    while (mon.mon_mbx.size < 100)
        @(dut.if.drv_cb);
    // Pull tx from mon_mbx and check correctness
end
endmodule
```
Conclusion

● SystemVerilog makes design easier and clearer vs Verilog
● SystemVerilog has many useful verification features not found in open-source testing environments
  ○ SVA, coverpoints, constrained random
● I’ve only scratched the surface
  ○ UVM
  ○ Hardware modeling
  ○ IPC (DPI/VPI)
● Play around: https://www.edaplayground.com/x/CK
References

https://verificationguide.com/systemverilog/systemverilog-tutorial/
https://www.chipverify.com/systemverilog/systemverilog-tutorial
https://www.doulos.com/knowhow/systemverilog/systemverilog-tutorials/systemverilog-assertions-tutorial/
https://www.systemverilog.io/sva-basics

Advanced notes on SystemVerilog covergroups: https://staging.doulos.com/media/1600/dvclub_austin.pdf
Notes on Vendor Support

- just mention the vendor support caveats verbally
Addendum Points

- Simulation loop scheduling, 4 state simulation
- x pessimism / optimism
- sources of mismatch between simulation and synthesis
- multiported memories and collision handling
- literals are 32 bits wide by default
- default_nettype
Tagged Unions

- too complicated a subject for this lecture