Q: *I need your advice in getting started with verification with systemverilog*
I 've built various projects using Verilog and they was FPGA and ASIC synthesizable. what I need now is a road map that get me started with verification for example SVA , UVM and SV testbenches

A: That's a very broad question because the answer requires knowledge of various disciplines that can be learned. It also requires the use of verification models whether it be a quick one-time-use model with a set of tests or reusable and extendable models with many sets of distinct tests. I'll provide with guidelines and I welcome comments from this community.

My general approach in writing a model (*e.g., code, document,)* is to use templates or existing models as a base. This simplifies the task significantly. It also is a good teaching tool. *A plug: in all my books, many now donated, http://systemverilog.us/vf/Cohen_Links_to_papers_books.pdf , I provide complete examples with annotations in my text and figures. I was criticized by some by not being "neat", but I find this approach more explanatory, with little concern to "pretty".*

Understand the basics of SystemVerilog Verification:
Given that you used SV in design and simple verification models, you have a general idea of the needed concepts. Understanding classes is a must. However, there are concepts and approaches that you need to consider as explained below.

Given a set of requirements, write a verification plan
A verification plan is a document that explains your verification approach (e.*g., transaction-based, reuse, tools, languages, models (IP), etc*) and what will be verified and covered.
My donated book *Component Design by Example … A step-step process using VHDL with UART as vehicle* https://rb.gy/9tcbhl specifies how to write: Requirement specification, Architectural plan, Verification plan, and Documentation and delivery procedures. This may be too formal and rigorous for some, but glancing at this approach will provide you with a model that you can construct to meet your needs and schedules. Your verification plan should also specify the use of assertions and coverage to address the level of testing and that requirements are met. Assertions are not necessarily SVA because assertions are just statements that what you present/state is correct. SVA can provide statements that your design requirement properties are correct. SVA can also provide functional coverage. SV also provides coverage of values/states. Tools provide automatic code coverage. Tools also provide timing analysis, metastability, fsm locking, etc. There are SO MANY things that you can put into a verification plan document!

The testbench
I see this as a broad question because designs go through several stages of evolutions and integrations. Generally, large designs are partitioned into smaller sections called *partitions*, each having its own interface. These partitions are then integrated into the larger design entity and tested at the functional level. Each designer, in coordination with a top-level verification engineer, is responsible to test his/her partition. I generally recommend a quick-and-dirty approach that progresses into a more complex verification model. My approach consists of:

1. **Assertions:** SVA with support logic if needed. These assertions can be written in-line with the code or in a checker/module bound to the design.
2. **Constrained-random tests:** This type of tests very quickly identifies errors in your assertions, your understanding of the requirements, and your model (*the early mortally issue)*.
   For this, I developed a template that I tune as needed. You can also add directed tests if needed.

```
module top;
  timeunit 1ns;  timeprecision 100ps;
  `include "uvm_macros.svh"   import uvm_pkg::*;
  bit clk, a, b, reset_n;
  default clocking @(posedge clk); endclocking
  initial forever #10 clk = !clk;
  initial begin $timeformat(-9, 0, " ns", 10);  $display("%t", $realtime); end
  // Assertions here
  // RTL model here: my_rtl my_rtl1(clk, a, b);

  initial begin    // The test vector generation
    $dumpfile("dump.vcd"); $dumpvars;
    bit v_a, v_b, v_err;
    repeat (200) begin
      @(posedge clk);
      if (!randomize(v_a, v_b, v_err) with {
        v_a   dist {1'b1 := 1, 1'b0 := 1};
        v_b   dist {1'b1 := 1, 1'b0 := 2};
        v_err dist {1'b1 := 1, 1'b0 := 15};
      }) `uvm_error("MYERR", "This is a randomize error");
      a <= v_a;
      if(v_err==0) b<=v_b; else b<=!v_b;
    end
    #20;
    $finish;
  end
endmodule
```

In this model, I use uvm for the UVM severity levels. Example of a severity level in SVA:

```
  string tID="UART ";
  default clocking def_cb @ (posedge clk);  endclocking : def_cb
  ap_LOW: assert property(a) else
    `uvm_info(tID,$sformatf("%m : error in a %b", a), UVM_LOW); // Line 9
  ap_MEDIUM: assert property(a) else
    `uvm_info(tID,$sformatf("%m : error in a %b", a), UVM_MEDIUM); // Line 11
  ap_HIGH: assert property(a) else
    `uvm_info(tID,$sformatf("%m : error in a %b", a), UVM_HIGH);   // Line 13
```

3. Transaction-based tests, quick-and-dirty

If you don't know UVM and want to get to something closer to it and fast, you can ease into transaction-based modeling in your partition without going UVM. This becomes a level of modeling at a higher level than a quick-and-dirty fast test. This also applies to models where transactions can be clearly identified, like READ, WRITE, PUSH, IDLE, etc. These transactions can be upgraded to UVM as a later stage of design integration.

My donated book (*$3, Amazon min charge*) *A Pragmatic Approach to VMM Adoption … a SV Framework for Testbenches 2007* is precursor to UVM and demonstrates by example the application of transaction-based modeling for a FIFO.

http://SystemVerilog.us/vf/VMM/VMM_pdf_release070506.zip

http://SystemVerilog.us/vf/VMM/VMM_code_release_071806.tar

For example, this code from that book can be modified by excluding the library references and used to generate transactions.

```
package fifo_pkg;
  timeunit 1ns; timeprecision 100ps;
  `define TOP fifo_tb
  typedef enum {PUSH, POP, PUSH_POP, IDLE, RESET} fifo_scen_e;
  typedef enum {PUSH_MODE, POP_MODE} mode_e;
  typedef enum {PASSED, FAILED} fifo_status_e;
  typedef enum {DONE_GEN, DONE_BFM} notification_e;
  parameter BIT_DEPTH = 4;
  parameter FULL = 16;
  parameter WIDTH = 32;
  typedef   logic [WIDTH-1 : 0] word_t;
  typedef   word_t [0 : (2**BIT_DEPTH-1)] buffer_t;
endpackage : fifo_pkg

class Fifo_xactn; //  extends vmm_data;
  rand fifo_scen_e kind;
  rand word_t  data;       // data to push
  rand bit [7:0] idle_cycles;
  rand bit [7:0] reset_cycles;
  time xactn_time;
  constraint cst_data {
      data  < 1024;  }
 constraint cst_idle {
   idle_cycles inside {[1:3]};  }
 constraint cst_reset {
   reset_cycles inside {[1:10]};  }

  constraint cst_xact_kind {
    kind dist {
      PUSH := 400,
      POP := 300,
      PUSH_POP :=200,
      IDLE := 30,
      RESET := 10
    };
  } // cst_xact_kind
endclass:Fifo_xactn

function string Fifo_xactn::psdisplay(string prefix);
    $sformat(psdisplay,
              "%s #%0d.%0d.%0d Fifo Xaction %s ",
              prefix, this.scenario_id, this.stream_id, this.data_id,
            this.kind.name());

    if (this.kind == IDLE)
      $sformat(psdisplay, "%s Cycles %0d ", psdisplay, this.idle_cycles);
    if (this.kind == RESET)
      $sformat(psdisplay, "%s Cycles %0d ", psdisplay, this.reset_cycles);
endfunction : psdisplay
```

You can then instantiate the class, randomize it, and do something like the following to call tasks to drive the DUT.

```
 case (fifo_xactn_0.kind)
        PUSH :
        begin
          this.push_task(fifo_xactn_0.data);
        end
        POP  :
        begin
         this.pop_task();
        End
```

…

4. Transaction-based tests, a la UVM

For this, I would start with a template or a previous working model and tune it to your needs. There are books and free material on UVM.   I don't consider myself an expert in this field as I concentrated on assertions and education. However, I strongly recommend my co-author Srinivasan Venkataramanan, CEO & Co-Founder at VerifWorks. He is an expert in many disciplines, from chip design (from start to release to fab), to many computing languages, to PSL, SVA, UVM, and tool construction.
https://www.linkedin.com/in/svenka3/ Also see his LinkedIn posts.
https://verifworks.com/ UVM worldwide consulting, Low Power UPF consulting Intellectual Properties (IP) primarily the verification oriented ones (VIP, Checker/Assertion IPs – CIP, Language testsuites etc.)

5. Learning SVA

There are many books and free material on SVA, including mine: *SVA Handbook 4th Edition,* 2016 ISBN 978-1518681448 https://rb.gy/4abc8v

You'll find that most assertions are simple in formation with a simple antecedent and a simple consequent (e.g., `@(posedge clk) $rose(req) |-> ##[1:5] ack;`). However, there are many concepts that need to be understood. Over the years, aside from updated versions of my book, I also wrote several papers on the subject. Below is a list of the papers; I recommend that you do study them as they go deep into the language and its nuance applications.

1	**Understanding the SVA Engine Using the Fork-Join Model** https://verificationacademy.com/verification-horizons/july-2020-volume-16-issue-2 Using a model, the paper addresses important concepts about attempts and threads. Emphasizes the total independence of attempts.
2	**Reflections on Users' Experiences with SVA, part 1** https://verificationacademy.com/verification-horizons/march-2022-volume-18-issue-1/reflections-on-users-experiences-with-systemverilog-assertions-sva Important concepts on EXPRESSING REQUIREMENTS, Terminology, threads in ranges and repeats in antecedents, multiple antecedents.
3	**Reflections on Users' Experiences with SVA, part 2** https://verificationacademy.com/verification-horizons/july-2022-volume-18-issue-2/reflections-on-users-experiences-with-sva-part-2 Addresses the usage of these four relationship operators: **`throughout`**, **`until`**, **`ntersect`**, **`implies`** operators
4	**Understanding Assertion Processing Within a Time Step** https://systemverilog.us/vf/Understanding_assertion_processing.pdf This paper goes into detail about how evaluation regions should be handled by a simulator as desc
5	**Understanding and Using Immediate Assertions** https://verificationacademy.com/verification-horizons/december-2022-volume-18-issue-3/understanding-and-using-immediate-assertions
6	**SVA Package: Dynamic and range delays and repeats** https://rb.gy/a89jlh Provides a library and model solutions
7	**SUPPORT LOGIC AND THE always PROPERTY** http://systemverilog.us/vf/support_logic_always.pdf Provides examples of support logic needed for certain types of requirements where the strict use of only SVA does not cover.
8	**SVA in a UVM Class-based Environmen** https://verificationacademy.com/verification-horizons/fFebruary-2013-volume-9-issue-1/SVA-in-a-UVM-Class-based-Environment Explains how SVA complements a UVM class-based environment. It also demonstrates how the UVM severity levels can be used in all SVA action blocks instead of the SystemVerilog native severity levels.