System Verilog (SV)

We'll look at

- FPGA / ASIC design challenges
- What is System Verilog?
- see how SV is used for design & syn.

Average growth rate of gate count → ≈ 25% per yr

RTL development is arguably the most time consuming activities in the design schedule

Ques main source of project delay?

Ans: spec changes

↓

re coding
new syn & new verification
new timing closure

Design reuse challenge

studies show >50% of IP is reused (internal (i.e. 2nd party))

difficulty in (next pg)

reading another designer's code

- engr no longer employed
- poor prior coding practices
  - poor documentation
  - { data structures & bus structures not intuitively encapsulated

basic building blocks of most designs ←

(Verilog 2001 ~~not~~ doesn't provide effective constructs)

Consequently,

✓ code for I/F and comm protocols must be re-designed

✓ common tasks/funcs and data abstractions not re-used efficiently

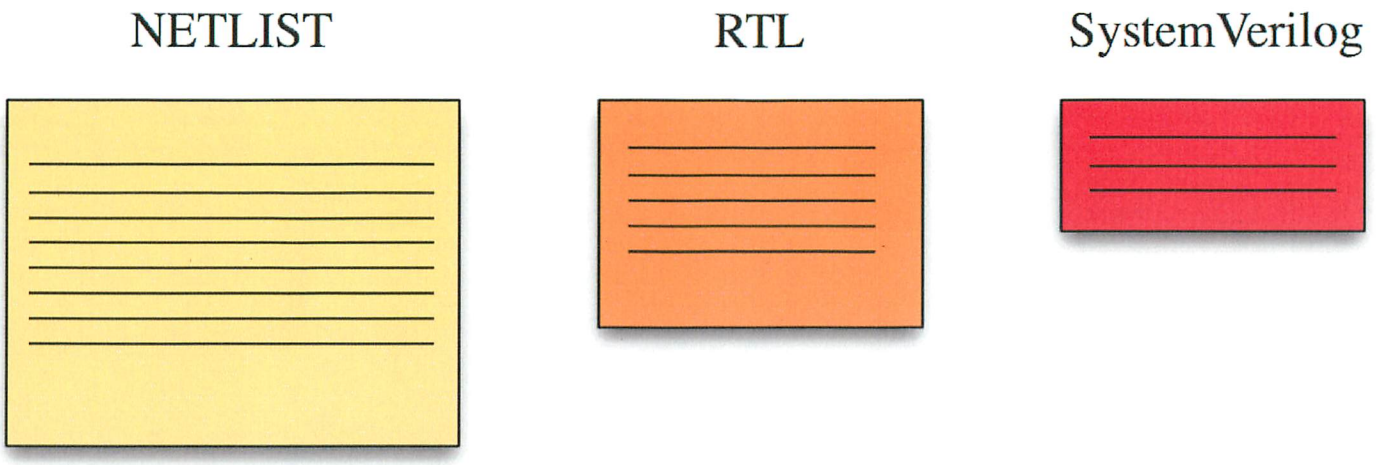
Verification challenge

- ≈ 25% of entire schedule
- simulation results ≠ synthesis results
- more code = more bugs
- more RTL code ⇒ more verification code

1800

- SV became an IEEE std in 2005
- superset of Verilog 2001
- raises the level of abstraction
- has RTL enhancements for improved conciseness and readability
- eases verification efforts
- used for both design and verification

has advanced testbench features



Reduction in RTL code size helps mitigate simulation and synthesis mismatch issues...

Notes

SV is not a new language

- it extends Verilog
- it remains an RTL Lang.

the enhances worranted a new name (hence, S Verilog)

Benefits for design

- conciseness

(show code fig again)

- Abstract data modelling
  easier to describe complex comms
- guards against simulation mismatches

e.g.

```
always_comb
    if(en) q <= d;
```

if synthesized as D-FF, tools will flag the inconsistency

Youtube
video
"SystVer as the new Verilog"

~~SystemC~~

Let's look at some SysV features

(1) with the exception of tri-state buses, 4-valued logic is rarely need (i.e. {0, 1, x, z})

So, SysV introduces 2-~~bit~~ state data types

table 1

(2) new abstract & user defined types
e.g. packed and unpacked structures

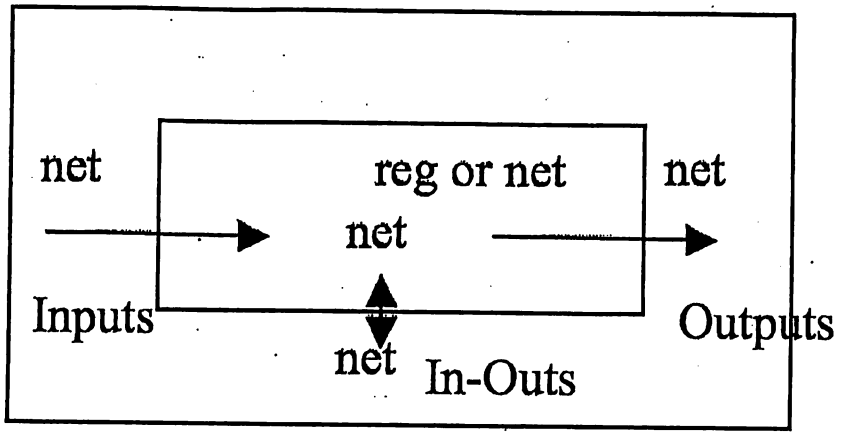
Two-state integer types {0,1}:

Type	Description	Example
bit	user-defined size	bit [3:0] a_nibble;
byte	8 bits, signed	byte a,b
shortint	16 bits, signed	shortint c,d;
int	32 bits, signed	int i,j;
longint	64 bits, signed	longint lword;

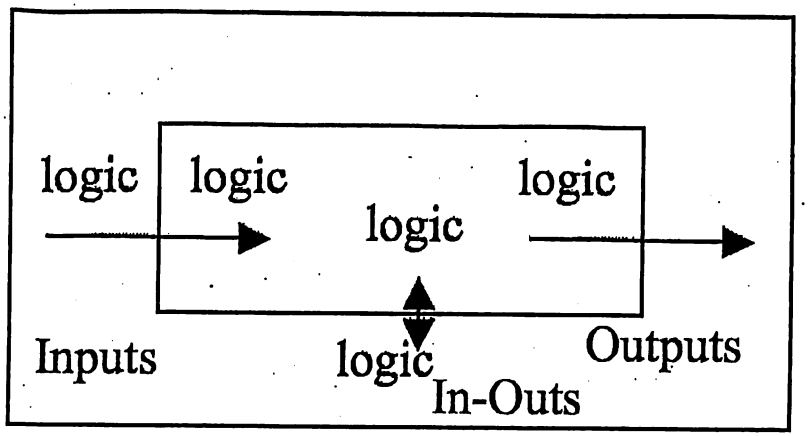
Four-state integer types {0,1,X,Z}:

Type	Description	Example
reg	user-defined size	reg [7:0] cmp;
logic	identical to "reg"	logic [7:0] cmp;
integer	32 bits, signed	integer j,k;

NOTE: "logic" is preferred over "reg"



Verilog Model
using reg or net



System Verilog Model
using logic

ABstract data Modelling

- user-defined types
- enumerated types
- structures
- I/F's

---

user-defined types

easy

you can extend name of nets & variables with user-defined names

can use typedef (like in C)

e.g.

```
typedef logic [7:0] instruction;
instruction instr1, instr2;
```

↑ ↑ these are 8-bit reg variables

easier to read ☺

easy

notice how this allows modeling functionality at higher levels

enumerated types

allows designer to assign a set of legal variable values

e.g. FSM

Verilog 2001

```
Parameter  wait=0,
           load=1,
           store=2;
reg [1:0] state;
```

SV

```
typedef enum reg [1:0]
  {wait, load, store} state;
```

arbitrary

note values for wait, load and store not reg'd

moreover, if anywhere in the code state assigned a value other than 'wait', 'load' or 'store' it would be flagged ☺

Notice how this make

- Hdwe models more concise & readable
- perfect for modelling FSM and instructions for ALUs

structures

allows modelling data at higher levels

e.g (data packets)

Verilog 2001

```
logic reg [31:0]   src_addr;
 "    reg [31:0]   dest_addr;
 "    reg [31:0]   instr_opcode;
 "    reg [31:0]   instr_payload;
```

no way to indicate these are part of a large data element (e.g. network packet)

SV

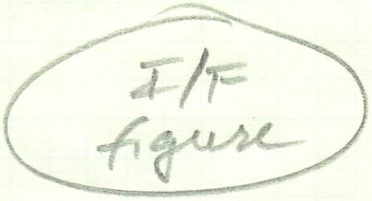
```
typedef struct packed {          ← note '{'
logic reg [31:0] src_addr,       ← not ';'
  "   reg [31:0] dest_addr,
  "   reg [31:0] instr_opcode,
  "   reg [31:0] instr_payload   } ← note nothing
      } pkt;
```

now, in module,

```
pkt  pkt_inst;
  :
if (pkt_inst.dest_addr == 32'hFFFO)
  :
```

ABstract data modelling : I/F

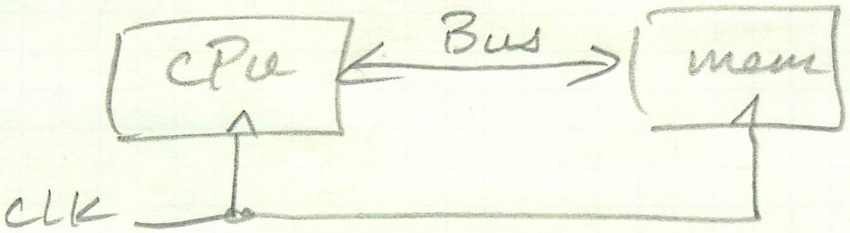
In Verilog 2001, bus layer is lost



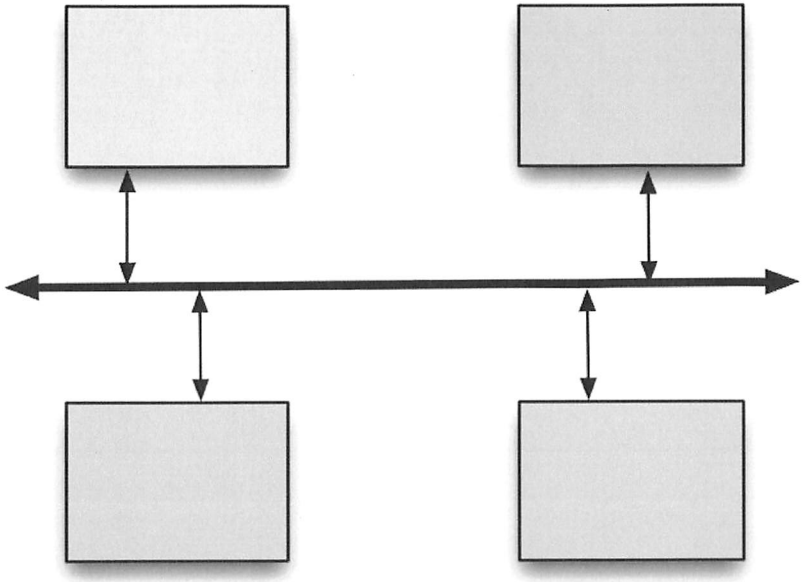
a rats nest across blocks
not obvious a bus exists
not readible
adding signals is error prone
↑
bus

SV introduced I/F
which encapsulates
- comms
- interconneativity

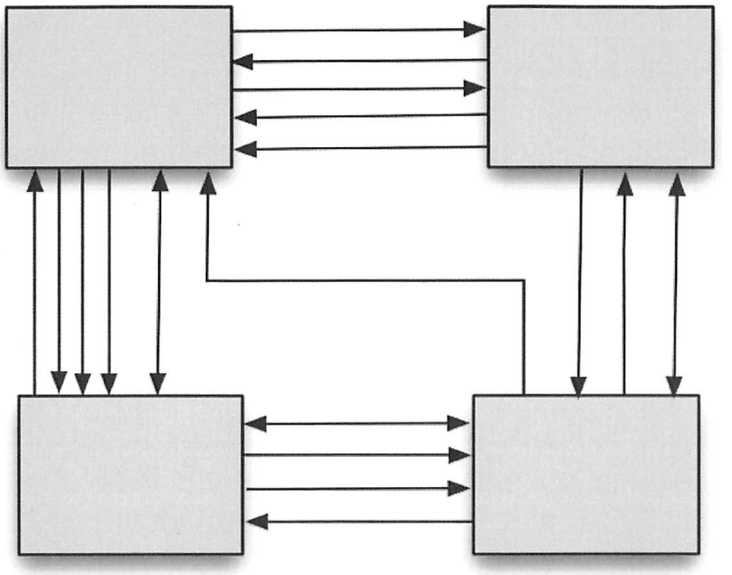
detailed example



Conceptual Design



Verilog 2001 Implementation



First thing is we define an I/F (almost like a module)

keyword ↙ ↙ name

```
interface bus;
```

signal def (but no direction) {

```
  reg clk, req, grant, start, rdy;
  reg [7:0] addr, data;
  reg [1:0] mode;
```

```
/* define protocols
   can use procedural blocks,
   continuous assignments,
   tasks, funcs, etc. */
```

task for common protocol →

```
task slvread;

// use modport to define directions
  modport cpu_port (inout data,
                    output addr,
                    ... );

  modport mem_port ( inout data,
                     input addr,
                     import slvread,
                     ... );

endinterface: bus
```

Modport defines directions
So how do we use this in the design?

```
module mem (bus.mem_port bus_mem);

    always @ (bus_mem.start)
        if (bus_mem.mode[0] = 1'b0)
            bus_mem.slvread)
        else
        ...
```

the dot notation allows access to all bus signals

New Operators

SystemVerilog adds a number of new operators, mostly borrowed from C. These include increment (++) and decrement (--), and assignment operators (+=, -=, ...). The *wild equality* operators (=== and !==) act like the comparisons in a casex statement, with X and Z values meaning "don't care".

New loop statements

Also from C is the *do-while* loop statement and *break* and *continue*. The new *foreach* loop is used with array variables. The *for* loop has been enhanced, so that the following is permitted:

```
for (int i = 15, logic j = 0 ; i > 0 ; i--, j = ~j) ...
```

note 2 conditions

Do loops next pg

Labelling

In Verilog, you may label *begin* and *fork* statements:

```
begin : a_label
```

In SystemVerilog the label may be repeated at the end:

```
end : a_label
```

This is useful for documenting the code. The label at the end must be the same as the one at the beginning. Modules, tasks and functions may also have their names repeated at the end:

```
module MyModule ...
...
endmodule : MyModule
```

do ... while Loops

- Verilog2001 has **`for`**, **`while`**, **`repeat`** and **`forever`** loops
- SystemVerilog adds the "C" style **`do...while`** loop
  - Syntax: **`do <statement(s)> while (<condition>);`**
  - It always executes once and the ***`condition`*** is checked after statement(s) execute.

```
initial begin
  integer i = 3;
  do begin
    $write("I:%d", i);
    if (i < 5)  $display(" is a Low Number");
    else  $display(" is a High Number");
    i++;
  end
  while (i<= 10);  // condition is a boolean expression
end
```

Synthesis Idioms

Verilog is very widely used for RTL synthesis, even though it wasn't designed as a synthesis language. It is very easy to write Verilog code that simulates correctly, and yet produces an incorrect design.For example, it is easy unintentionally to infer transparent latches. One of the ways in which SystemVerilog addresses this is through the introduction of new *always* keywords: *always_comb*, *always_latch* and *always_ff*. *always_comb* is used to describe combinational logic. It implicitly creates a complete sensitivity list by looking at the variables and nets that are read in the process, just like *always* @* in Verilog-2001.

```
always_comb
  if (sel)
    f = x;
  else
    f = y;
```

so no sensitivity list need for comb logic

```
always_latch // inferred sensitivity list
  if(en)
    q<=d;
```

n addition to creating a complete sensitivity list automatically, it recursively looks into function bodies and inserts any other necessary signals into the sensitivity list. It also is defined to enforce at least some of the rules for combinational logic, and it can be used as a hint (particularly by synthesis tools) to apply more rigorous synthesis style checks.Finally, always_comb adds new semantics: it implicitly puts its sensitivity list at the end of the process, so that it is evaluated just once at time zero and therefore all its outputs take up appropriate values before simulation time starts to progress.

always_latch and *always_ff* are used for infering transparent latches and flip-flops respectively. Here is an example of *always_ff*:

```
always_ff @(posedge clock iff reset == 0 or posedge reset)
  if (reset)
    q <= 0;
  else if (enable)
    q++;
```

if and only if

posedge clock trigger (i.e. is an event)
if and only if it occurs when reset == 0

The advantage of using all these new styles of always is that the synthesis tool can check the design intent.

Unique and Priority

Another common mistake in RTL Verilog is the misuse of the *parallel_case* and *full_case* pragmas. The problems arises because these are ignored as comments by simulators, but they are used to direct synthesis. SystemVerilog addresses this with two new keywords: *priority* and *unique*.

Unlike the pragmas, these keywords apply to *if* statements as well as *case* statements. Each imposes specific simulation behaviour that is readily mapped to synthesised hardware. *unique* enforces completeness and uniqueness of the conditional; in other words, exactly one branch of the conditional should be taken at run-time. If the specific conditions that pertain at run-time would allow more than one branch of the conditional, or no branch at all, to be taken, there is a run-time error. For example, it is acceptable for the selectors in a case statement to overlap, but if that overlap condition is detected at runtime then it is an error. Similarly it is okay to have a unique case statement with no default branch, or an if statement with no else branch, but at run time the simulator will check that some branch is indeed taken. Synthesis tools can use this information, rather as they might a full_case directive, to infer that no latches should be created.

priority enforces a somewhat less rigorous set of checks, checking only that at least one branch of the conditional is taken. It therefore allows the possibility that more than one branch of the conditional could be taken at run-time. It licenses synthesis to create more extravagant priority logic in such a situation.

Prev Next

SystemVerilog Assertions Tutorial

Introduction

Assertions are primarily used to validate the behaviour of a design. ("Is it working correctly?") They may also be used to provide functional coverage information for a design ("How good is the test?").
Assertions can be checked dynamically by simulation, or statically by a separate property checker tool – i.e. a formal verification tool that proves whether or not a design meets its specification. Such tools may require certain assumptions about the design's behaviour to be specified.
In SystemVerilog there are two kinds of assertions: immediate (`assert`) and concurrent (`assert property`). Coverage statements (`cover property`) are concurrent and have the same syntax as concurrent assertions, as do `assume property` statements. Another similar statement – `expect` – is used in testbenches; it is a procedural statement that checks that some specified activity occurs. The three types of concurrent assertion statement and the `expect` statement make use of sequences and properties that describe the design's temporal behaviour – i.e. behaviour over time, as defined by one or more clocks.

Immediate Assertions

Immediate assertions are procedural statements and are mainly used in simulation. An assertion is basically a statement that something must be true, similar to the `if` statement. The difference is that an `if` statement does not assert that an expression is true, it simply checks that it is true, e.g.:

```
if (A == B) ...  // Simply checks if A equals B
assert (A == B); // Asserts that A equals B; if not, an error is generated
```

If the conditional expression of the immediate `assert` evaluates to X, Z or 0, then the assertion fails and the simulator writes an error message. An immediate assertion may include a pass statement and/or a fail statement. In our example the *pass statement* is omitted, so no action is taken when the `assert` expression is true. If the pass statement exists:

```
assert (A == B) $display ("OK. A equals B");
```

it is executed immediately after the evaluation of the `assert` expression. The statement associated with an `else` is called a *fail statement* and is executed if the assertion fails:

```
assert (A == B) $display ("OK. A equals B");
  else $error("It's gone wrong");
```

Note that you can omit the pass statement and still have a fail statement:


```
assert (A == B) else $error("It's gone wrong");
```

The failure of an assertion has a severity associated with it. There are three severity system tasks that can be include specify a severity level: `$fatal`, `$error` (the default severity) and `$warning`. In addition, the system task `$info` failure carries no specific severity.
Here are some examples:

```
ReadCheck: assert (data == correct_data)
             else $error("memory read error");
Igt10: assert (I > 10)
         else $warning("I has exceeded 10");
```

The pass and fail statements can be any legal SystemVerilog procedural statement. They can be used, for example, t an error flag, increment a count of errors, or signal a failure to another part of the testbench.

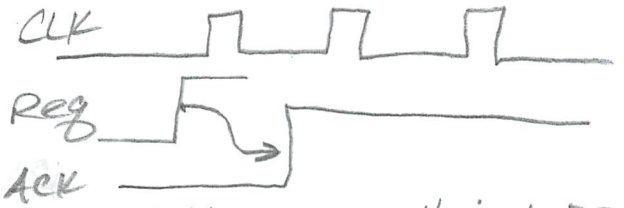
```
AeqB: assert (a == b)
  else begin error_count++; $error("A should equal B"); end
```

Concurrent Assertions

The behaviour of a design may be specified using statements similar to these:
"The Read and Write signals should never be asserted together."
"A Request should be followed by an Acknowledge occurring no more than two clocks after the Request is asserted."
Concurrent assertions are used to check behaviour such as this. These are statements that assert that *specified properties* must be true. For example,

```
assert property (!(Read && Write));
```

asserts that the expression `Read && Write` is never true at any point during simulation.



Properties are built using sequences. For example,

```
assert property (@(posedge Clock) Req |-> ##[1:2] Ack);
```

after request asserted ack follows within 1->2 clocks

where `Req` is a simple sequence (it's just a boolean expression) and `##[1:2] Ack` is a more complex sequence expression, meaning that `Ack` is true on the next clock, or on the one following (or both). `|->` is the implication operator, so this assertion checks that whenever `Req` is asserted, `Ack` must be asserted on the next clock, or the following clock.
Concurrent assertions like these are checked throughout simulation. They usually appear outside any initial or always blocks in modules, interfaces and programs. (Concurrent assertions may also be used as statements in initial or always blocks. A concurrent assertion in an initial block is only tested on the first clock tick.)
The first assertion example above does not contain a clock. Therefore it is checked at every point in the simulation. The second assertion is only checked when a rising clock edge has occurred; the values of `Req` and `Ack` are sampled on the rising edge of Clock.