SystemVerilog Series

Getting Started: A Crash Course

@nebu

The Title Was a Lie

- In this presentation, we'll cover the subset of SystemVerilog that's used for RTL design (and not for verification).
- This is (almost) pure Verilog-2005.

Prerequisite Background

- ECE120/CS233-level of digital electronics understanding. If you don't know most of the terms: logic gate, combinational logic, critical path, flip-flop, latch, FSMs, this is probably not a good place to start.
- SIGARCH has meetings that introduce basic digital electronics, that material serves as a prerequisite to these slides.

Modules and Ports

- A (digital) circuit can be thought of as a bunch of black boxes connected together with wires.
- Verilog models this using “modules”, which are connected using “ports”.
- TODO: insert diagram.

Here’s a Module

```
module my_module_name (
  input  a,
  input  b,
  output x,
  output y
);

  // Module contents

endmodule : my_module_name
```

Buses

```
module my_module_name (
  input  [3:0] a,
  input        b,
  output       x,
  output [0:7] y  // unorthodox
);

  // Module contents

endmodule : my_module_name
```

Questions?

OK, Model This Module Structure in Verilog

Answer: Module A

```
module module_a (
  input        clk,
  input        rst,
  input        ready,
  output       valid,
  output [3:0] data
);

endmodule : module_a
```

Answer: Module B

```
module module_b (
  input        clk,
  input        rst,
  input        valid,
  input [3:0]  data,
  output       ready
);

endmodule : module_b
```

Nesting/Module Instantiation

Modules typically form hierarchies: modules can *instantiate* other modules. Here's a top level module instantiating A and B, and hooking them up.

```
module top;
    logic       clk;
    logic       rst;
    logic [3:0] data;  // This is how module-local signals
    logic       ready; // are declared.
    logic       valid;

// Continued...
```

Module Instantiation

```
    module_a module_a_instance_name (
      .clk   (clk),
      .rst   (rst),
      .data  (data),
      .ready (ready),
      .valid (valid)
    );

    module_b module_b_i (
      .clk   (clk),
      .rst   (rst),
      .data  (data),
      .ready (ready),
      .valid (valid)
    );

endmodule : top
```

Questions?

Writing a Flip-Flop

```
module dff (
  input        clk,
  input        rst,
  input        d,
  output logic q
);

  always @(posedge clk) begin // TODO: introduce always_ff later
    if (rst == 1'b1) begin
      q <= 1'b0;
    end else begin
      q <= d;
    end
  end

endmodule : dff
```

What is `always` anyway?

- Remember, Verilog is a hardware *description* language.
- But, it was originally designed for simulating hardware.
- This is why `always` blocks have sensitivity lists: the `@` tells the `always` block when to “trigger”.
- Once triggered, the statements inside the `always` block’s `begin` / `end` are executed.

Can you implement a register?

- 16 bits wide.
- Has an enable signal.
- Bonus: has an asynchronous active-low reset.
- Bonus: is parameterizable.

16-bit register

```
module register (
  input               clk,
  input               rst_n,
  input        [15:0] data_in,
  input               enable,
  output logic [15:0] data_out
);

  always_ff @(posedge clk) begin
    if (rst == 1'b1) begin
      data_in <= 16'b0;
    end else begin
      if (enable == 1'b1) data_out <= data_in;
    end
  end

endmodule : register
```

Let’s take a break and do some combinational logic: a MUX.

```
module mux (
  input        a,
  input        b,
  input        sel,
  output logic r
);

  always @(a or b) begin // Explicit sensitivity list. Bad.
    if (sel == 1'b0) r = a;
    else r = b;
  end

endmodule : mux
```

Use always_comb.

```
module mux (
  input        a,
  input        b,
  input        sel,
  output logic r
);

  always_comb begin
    if (sel == 1'b0) r = a;
    else r = b;
  end

endmodule : mux
```

Here’s another possible combinational block

```
always_comb begin
  and_of_buses          = a & b;
  or_of_buses           = a | b;
  xor_of_buses          = a ^ b;
  addition_of_buses     = a + b;
  shifting              = a >> 4;
  unary_or_on_bus       = |a;
  weird_logic           = ~(a & b) | c;
  part_select           = a[7:4];
  concatenation         = {a, b, c};
  replication           = {4{a[7]}};
end
```

And, uh.

```
always_comb begin
  never_do_this[31:0] = {a[7-:5], &b, {16{c[1]}} ^ d};
end
```

Questions?

Can you implement a 3:1 MUX?

```
module mux3to1 (
  input        [7:0] a,
  input        [7:0] b,
  input        [7:0] c,
  input        [1:0] sel,
  output logic [7:0] q
);

endmodule : mux3to1
```

Solution

```
module mux3to1 (
  input        [7:0] a,
  input        [7:0] b,
  input        [7:0] c,
  input        [1:0] sel,
  output logic [7:0] q
);
  always_comb begin
    unique case (sel)
      2'b00: q = a;
      2'b01: q = b;
      2'b10: q = c;
      default: q = 'x;
    endcase
  end
endmodule : mux3to1
```

Can you implement...a shift register?

- 16 bits, synchronous active-high reset.
- On every clock, if `shift_en` is high, shift in a new bit, shift out the old bit.
- Parallel input on `load_en`.
- Parallel output.

Much easier with an interface!

Specifications are hard to write: think about edge cases.

```
module shift_regsiter (
  input                clk,
  input                rst,
  input         [15:0] data_in,
  input                shift_in,
  input                shift_en,
  input                load_en,
  output logic         shifted_out,
  output logic  [15:0] data_out
);

endmodule
```

Answer

- Note how this is completely independent of the width of the register `data_out`.
- Note how the edge case of `(shift_en && load_en)` is handled.

```
always_ff @(posedge clk) begin
  if (rst = 1'b1) data_out <= '0;
  else begin
    if (shift_en == 1'b1) begin
      {shifted_out, data_out} <= {data_out, shift_in};
    end
    if (load_en == 1'b1) data_out <= data_in;
  end
end
```