scientific analog

Digital Hardware Modeling with SystemVerilog

Scientific Analog, Inc.

July 2017

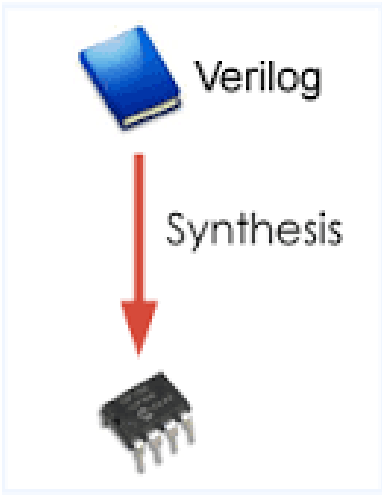
Overview

- *XMODEL* empowers SystemVerilog with fast and accurate modeling and simulation of ***analog circuits***
- Still, ***digital circuits*** are modeled and verified with SystemVerilog, leveraging the established simulators and verification flows (e.g. UVM)
  - And synthesis tools can convert the verified designs into gate-level implementation
- This lecture covers the basics of modeling digital systems and writing their testbenches in SystemVerilog

About Verilog / SystemVerilog

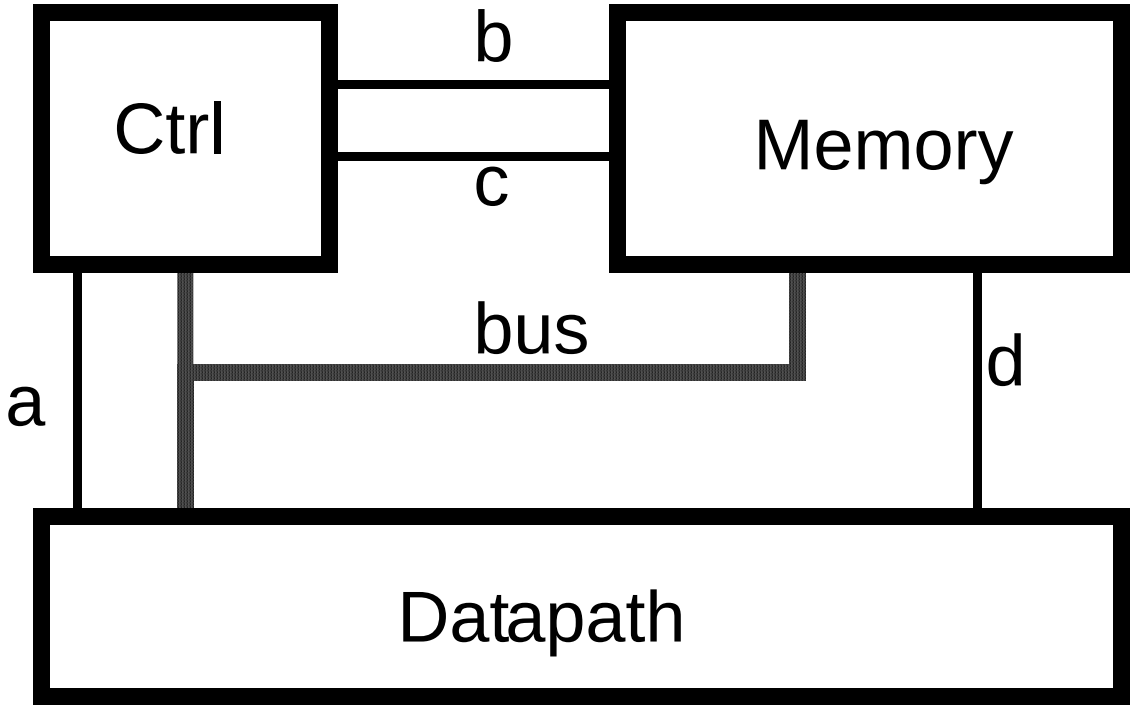
- Verilog language describes how a digital hardware system behaves (HDL)
  - Describes either what it is made of (***structural description***) or what is does (***functional description***)
  - Logic synthesis tools (e.g. Design Compiler) can translate Verilog models into gate-level implementation

- SystemVerilog is an extension to facilitate verification (e.g. writing testbenches)



Verilog View of the World

- A design consists of a set of modules communicating via signals (a.k.a. *signal-flow system models*)
  - Each module operates concurrently with the others



Specifying Functionalities

- Verilog gives you four ways to specify the model's functionalities:

Structural

- Make the function out of smaller blocks

Declarative

- Assign a function of the input to an output

Procedural

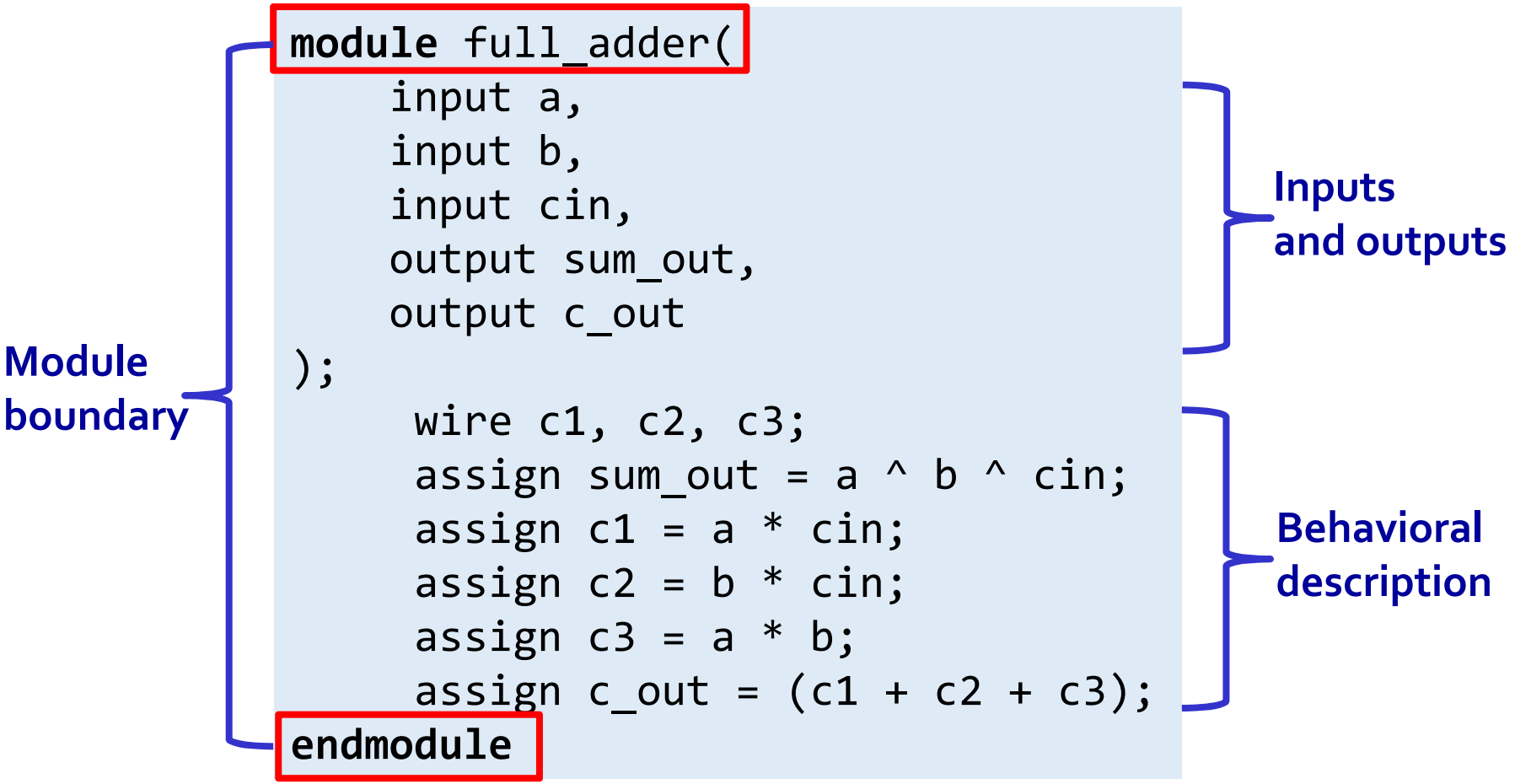
- Run a small code segment to describe the output

Functional

- May use non-synthesizable codes (e.g. analog models)

A First Look on Verilog

- Module is the basic unit of hierarchy in Verilog



Declaring Module Ports

- There are two styles:

  1. Listing the I/O and data type of each port along with its name (ANSI-style)

```
module D_FF ( output reg q,
              input d, clk, reset );
```

  2. Listing only the names first and define the I/O type later

```
module D_FF ( q, d, clk, reset );
output reg q;
input d, clk, reset;
```

- We recommend the first style

Data Types in Verilog

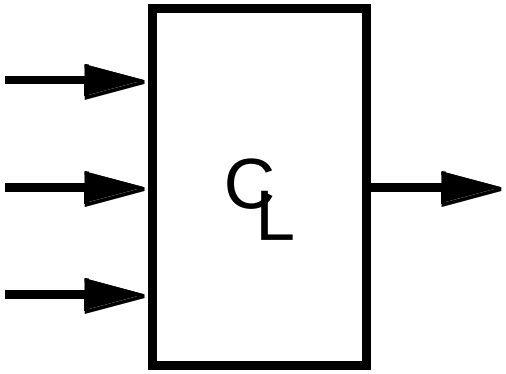
- The basic data types are ***wire*** and ***reg***
  - Both are for 4-value logic signals: 0, 1, X, and Z
  - X is for "don't know" or "don't care"
  - Z is for high impedance (floating; no driver state)
  - The difference between ***wire*** and ***reg*** will be discussed soon

- SystemVerilog and *XMODEL* add more data types
  - SystemVerilog: *real*, *integer*, *bit*, *logic* (=*reg*), …
  - XMODEL: ***xbit*** and ***xreal***

Describe Logic Using *assign*

- Describes a combinational logic between in & out

```
assign    nor = ~(b | c);
assign    a = x & y, o = x | y;
assign    sum[4:0] = a[3:0] + b[3:0];
assign    out = (sel) ? in1: in2;
```



- Called **"*continuous assignment*"**
  - You can use a C-like expression
  - Output is re-evaluated as soon as the inputs are updated
- Output variable must be a ***wire*** type

Exercise #1

- Write a full adder model in Verilog:

$$Sout = A \oplus B \oplus Cin$$

$$Cout = A \cdot B + B \cdot Cin + Cin \cdot A$$

- Hints:
  - The bitwise AND and OR operators are **&** and **|**, respectively
  - XOR operator ⊕ in Verilog is **^**
  - Start with a skeleton code in **models/full_adder.sv**

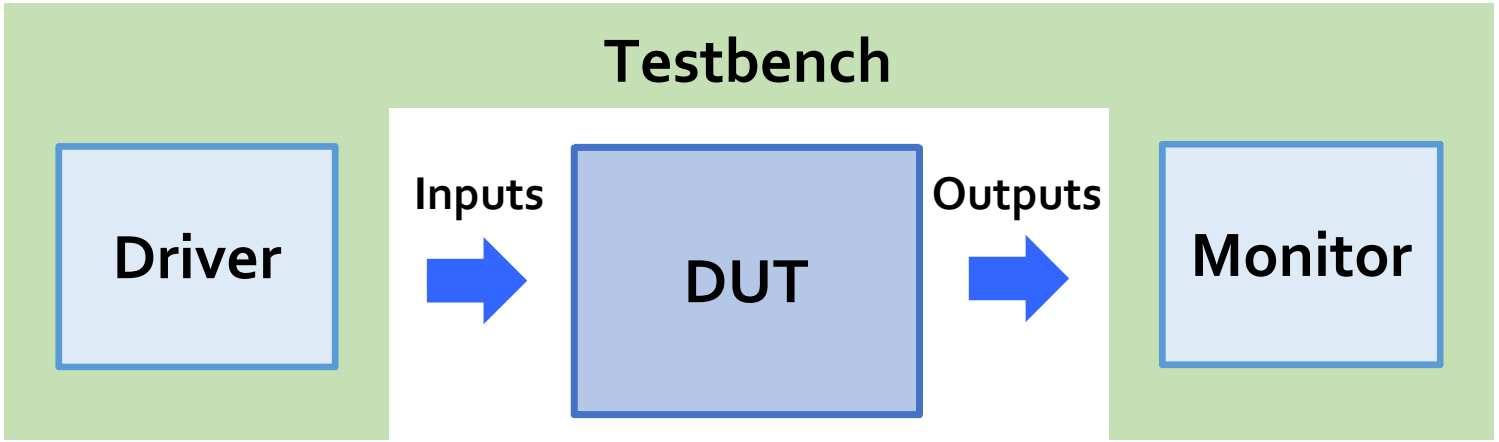
Answer #1

- Located in **models/answer/full_adder.sv**
- Many alternate answers are possible!

```
module full_adder (
    output Sout, Cout,
    input A, B, Cin
);

assign   Sout = A ^ B ^ Cin;
assign   Cout = (A & B) | (B & Cin) | (Cin & A);

endmodule
```

Testbench Writing

- You need a testbench module to verify your model ("device under test; DUT")
  - Testbench feeds the input signals to the DUT and observe if the outputs of DUT are correct
  - Testbench code need not be synthesizable (not hardware)



Initial Block

- This is a special type of procedural block
  - It is run just once, when the simulation starts
  - Does not need an activation list

```
initial begin
    clk = 0;
    in = 1;
end
```

- Useful when writing testbenches
  - e.g. sequencing signals into the modules under test
  - e.g. initializing simulation environment
- However, initial blocks do NOT turn into real hardware (after synthesis)
  - Real hardware must have explicit reset mechanisms
  - Best to use initial blocks ***only*** for non-hardware statements

Delays in Verilog

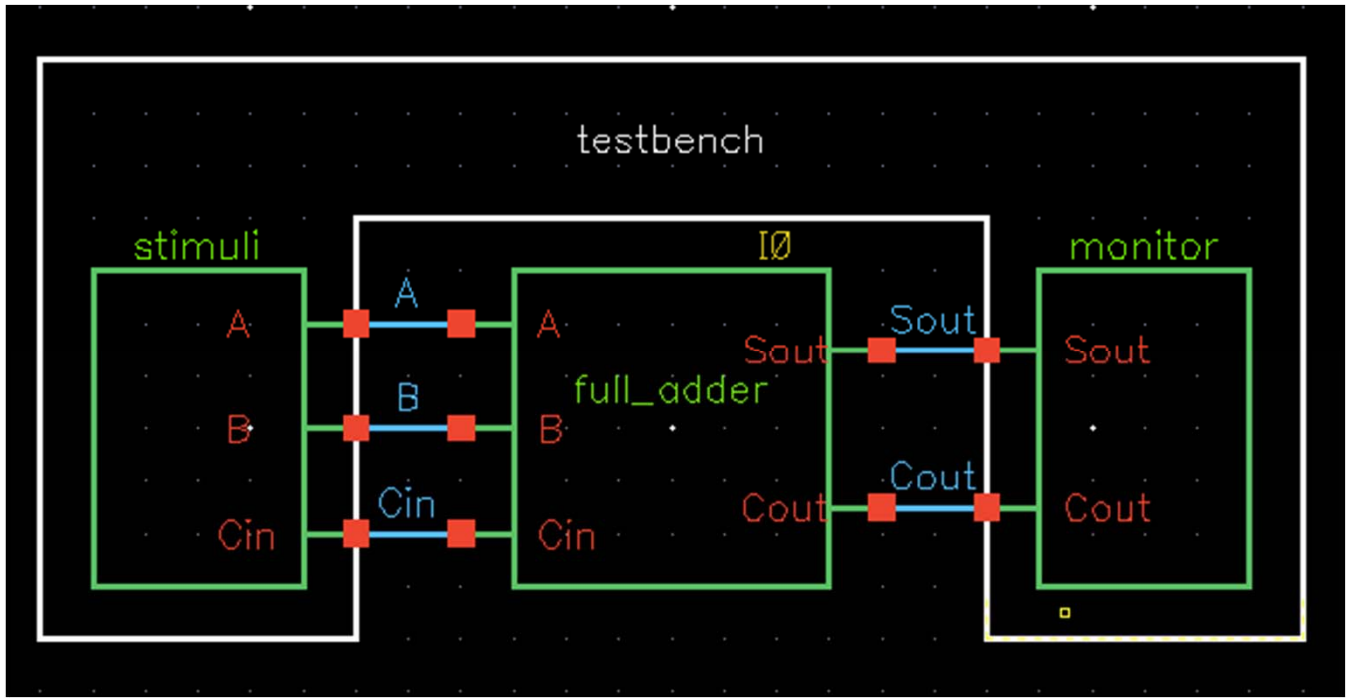
- ***# DelayAmount***
  - Whenever the simulator sees this symbol, it will pause for *DelayAmount* measured in simulation timeunits ("ticks")
  - The timeunit is defined by `timescale statement

    e.g. `timescale 1ps/1ps
  - In SystemVerilog, you can also specify absolute time delay

    e.g. #(10ns)
- Verilog simulators are very efficient in handling delays
  - Simply schedule the event in a future time and resume the operation then (the "event-driven" simulation)

Exercise #2: Simplest Testbench

- Explain what **sim/tb_full_adder/tb_full_adder.sv** does to verify the full adder model
- Run the simulation – can it fully verify the DUT?



Exercise #2: Simplest Testbench (2)

```
module tb_full_adder ();

    // signal declarations
    reg A, B, Cin;
    wire Cout, Sout;

    // stimuli generation and output monitoring
    initial begin
        A = 1;
        B = 0;
        Cin = 1;

        $display("A    = ", A);
        $display("B    = ", B);
        $display("Cin  = ", Cin);

        #10;
        $display("Sout = ", Sout);
        $display("Cout = ", Cout);
    end

    // instance declarations
    full_adder      DUT (.A(A), .B(B), .Cin(Cin), .Sout(Sout), .Cout(Cout));

endmodule
```

← Input value assignment

← Output value display

Procedural Description: *always*

- Runs a little program whenever a certain event occurs

```
always @(sel1 or in1 or in2) begin
       if (sel1)    out <= in1;
       else         out <= in2;
end
```

  - Internally each program executes sequentially
  - But all the different programs run concurrently
- Inside an always block, you can use control flow statements like if/else, for/while, case, etc.
- Output variable must be a ***reg*** type

Always Activation List

- Three forms of activation list for *always*:

  @(signal_name or signal_name or …)

  - Evaluate this block when any of the named signals changes

  @(*), or @*

  - Have Verilog figure out the right signals to trigger on

  @(posedge signal_name) or
  @(negedge signal_name)

  - Evaluates only on one edge of a signal
  - Makes an edge-triggered FF

Exercise #3

- Check out the source files in **sim/tb_full_adder_2**
- Does it do a better job of verifying the full adder?

Stimulus Driver

```
module stm_full_adder_2 (
    output reg A, B, Cin
);

always begin
    A = $urandom_range(1);
    B = $urandom_range(1);
    Cin = $urandom_range(1);

    $display("A    = ", A);
    $display("B    = ", B);
    $display("Cin  = ", Cin);

    #10;
end

endmodule
```

Periodically updates A, B, Cin with random values

Output Monitor

```
module mon_full_adder_2 (
    input Sout, Cout
);

always @(Sout or Cout) begin
    $display("Sout = ", Sout);
    $display("Cout = ", Cout);
end

endmodule
```

Records the outputs whenever they change

Random Stimuli Generation

- One common way to increase the verification coverage is to generate random stimuli signals
  - In fact, SystemVerilog has extensive supports for random signal generation and assertion checks

- A simple way to generate a random integer within a range is:

```
<variable> = $urandom_range( <max_value>, [<min_value>]);
```

  - The min_value assumes o if it is not specified

Flip-flop and Latch in Verilog

- Flip-flop:

```
always @(posedge clk) begin
      out <= in;
end
```

- D-Latch:

```
always @(clk or in) begin
      if (clk) out <= in;
end
```

- Note that '***always***' creates a storage if the output has a need to remember its previous value (hence '***reg***')

If-Else Statement

- Decide which code to run based on a condition
  - Syntax: **if (*condition*)**
    statement executed when True
    else
    statement executed when False
  - Example: a resettable FF:

```
always @(posedge clk or posedge reset) begin
       if (reset) out <= 1'b0;
       else out <= in;
end
```

Case Statement

- Useful when you have too many conditions to test:
- Example: A one-hot to binary encoder

```
always @(in) begin
     case(in):
          4'b0001: out = 2'b00;
          4'b0010: out = 2'b01;
          4'b0100: out = 2'b10;
          4'b1000: out = 2'b11;
          default: out = 2'bxx;
     endcase
end
```

For Statement

- **"for"** loop : repeat some statements while incrementing an index variable
- Example: A binary to one-hot decoder

```
module decoder ( output reg [3:0] out,
                 input [1:0] in );
    integer i;
    always @(in) begin
        for (i=0; i<4; i=i+1) begin
            out[i] = (in == i) ? 1'b1 : 1'b0;
        end
    end
endmodule
```

Exercise #4

- Describe a 4-bit counter, whose output ***count*** increments or decrements whenever clock ***clk*** rises

  count increments when **up_dnb** is 1
  count decrements when **up_dnb** is 0

- Hint:
  - Start with a skeleton code in **models/counter.sv**
  - Use the testbench in **sim/tb_counter** to check its correctness

Answer #4

- Located in **models/answer/counter.sv**

```
module counter (
    output reg [3:0] count,
    input up_dnb,
    input clk
);
initial count = 0;
always @(posedge clk) begin
    if (up_dnb) count <= count + 1;
    else count <= count - 1;
end
endmodule
```

Good Practice in Verilog Modeling

- Don't mix comb. and seq. logic in same always block!
  - Logic is much clearer
  - Avoids weird issues with resets
  - Avoids latch/flop inferring that you did not plan

Bad1: mixing comb. and seq. logic

```
always @posedge Clk
    counter <= counter +1
```

Bad2: unintended latch inferred

```
always @ signal_a
    if (signal_a)
        signal_b = signal_a + 1;
```

Good1:

```
always @*
    nx_counter = counter + 1

always @posedge Clk
    counter <= nx_counter
```

Good2:

```
always @ signal_a
    if (signal_a)
        signal_b = signal_a + 1;
    else
        signal_b = signal_a + 2;
```

Verilog Variables: *wire* and *reg*

- There are two types of variables in Verilog:
  - **wire** (all outputs of **assign statements** must be ***wire***)
  - **reg** (all outputs of **always blocks** must be ***reg***)
  - Note: bits in SystemVerilog are essentially two-valued reg's
- Both are 4-value logic variables
  - 1, 0, Z (high impedance), X (unknown)
- If you are describing a synthesizable digital block, you must use these two signal types only

Verilog Variables: *wire* and *reg* (2)

- Both ***wire*** and ***reg*** can be used as inputs anywhere
  - Inputs (RHS) in assign statements

```
assign bus = inA + inB;
```

  - Inputs (RHS) in always blocks

```
always @(in or clk)
    if (clk) out = in;
    // in can be a wire, out must be a reg
```

- The outputs of a module can also be either ***reg***'s or ***wire***'s.

Size of Variables

- In programming, you don't care much about size of the variables (except for large data structures)

- But in hardware it really matters
  - Need to know size of integers you want to add
  - So Verilog uses explicit sizes for variables
  - Even constants are specified with bit width
    - 3'b101 (3 bits of "101")
    - 4'b101 (4 bits of "0101" where the preceding 0 is omitted)

Using Vectors and Arrays

- An array of signals can be defined like this:

```
reg   [15:0]   a;
wire  [0:7]    b;
xbit  [3:0]    c, d, e;
```

- And the array elements can be indexed like this:

```
x = a[3]+a[2]+a[1]+a[0];     // indexing individual elements
y = b[0:3]+b[4:7];           // indexing range of elements
z = {c[1:0],d[2:1],e[3:2]}; // concatenation
```

- Further reading: check out *packed* vs. *unpacked* arrays

Additional Data Types: *int* and *real*

- Verilog supports data types other than ***reg*** and ***wire***
  - For instance, ***int*** and ***real*** express integer and floating-point numbers, respectively
  - These are for functional models only and can't be passed across the module boundary via ports

- SystemVerilog supports more data types and even allows passing them through ports
  - e.g. ***string***, ***struct*** and ***union*** in addition to ***int*** and ***real***
  - *XMODEL* uses ***struct*** to define ***xbit*** and ***xreal***

Exercise #5

- Write a 4-bit adder that adds two 4-bit inputs ***A*** and ***B*** and produces 4-bit output sum ***S*** and carry out ***C***

- Hint:
  - Start with a skeleton code in **models/add4.sv**
  - You may use ***assign*** or ***always*** statements

Answer #5

- Located in **models/answer/add4.sv**
- Many other answers are also possible!

```
module add4 (
    input [3:0] A, B,
    output [3:0] S,
    output C
);
wire    [4:0] sum;
assign  sum = A + B;
assign  S = sum[3:0];
assign  C = sum[4];
endmodule
```

Self-Checking Monitors

- Testbench in **sim/tb_add4**
- Checks whether the results are correct or not

Output Monitor

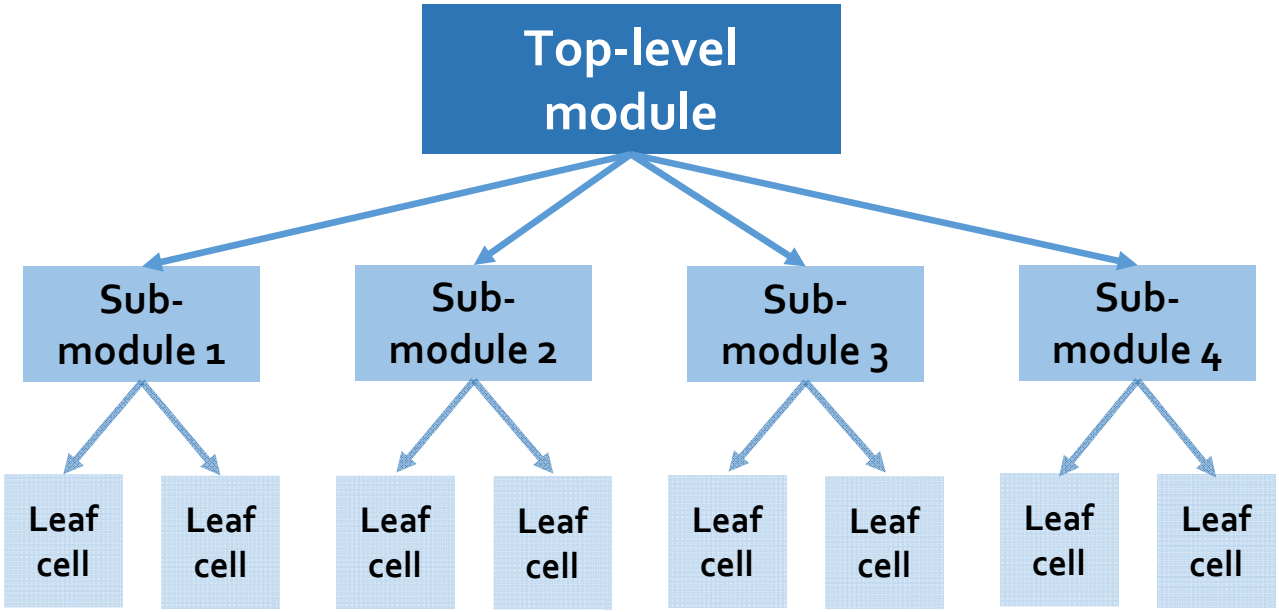
```
wire [4:0] sum;
assign sum = {C, S[3:0]};

always @(negedge clk) begin
    if (sum == A + B)
        $display("CORRECT: A=%b, ", A, "B=%b: ",
         B, "S=%b, ", S, "C=%b", C);
    else
        $display("WRONG  : A=%b, ", A, "B=%b: ",
         B, "S=%b, ", S, "C=%b", C);
end
```

Output Log

```
# WRONG  : A=xxxx, B=xxxx: S=xxxx, C=x
# CORRECT: A=0100, B=1101: S=0001, C=1
# CORRECT: A=1101, B=0110: S=0011, C=1
# CORRECT: A=0011, B=0110: S=1001, C=0
# CORRECT: A=1011, B=1110: S=1001, C=1
# CORRECT: A=1110, B=0001: S=1111, C=0
# CORRECT: A=0001, B=1000: S=1001, C=0
# CORRECT: A=0010, B=1001: S=1011, C=0
# CORRECT: A=1000, B=1010: S=0010, C=1
# CORRECT: A=0000, B=1001: S=1001, C=0
# CORRECT: A=1111, B=1111: S=1110, C=1
# CORRECT: A=0111, B=0100: S=1011, C=0
# CORRECT: A=0111, B=1001: S=0000, C=1
# CORRECT: A=0010, B=0001: S=0011, C=0
# CORRECT: A=1111, B=0001: S=0000, C=1
# CORRECT: A=1100, B=0010: S=1110, C=0
# CORRECT: A=0111, B=0110: S=1101, C=0
# CORRECT: A=0101, B=1111: S=0100, C=1
# CORRECT: A=1111, B=0011: S=0010, C=1
# CORRECT: A=0100, B=0111: S=1011, C=0
# CORRECT: A=1001, B=1010: S=0011, C=1
```

Hierarchical/Structural Modeling

- You can build a large system by putting smaller blocks together
- Use hierarchy/modularity to manage complexity



Structural Description

- You can instantiate a module in another module

```
module adder_set();

  wire a1, b1, c1;
  wire s_out1, c_out1;
  full_adder FA1(.a(a1), .b(b1), .cin(c1),
                 .sum_out(s_out1), .c_out(c_out1));

  wire a2, b2;
  wire s_out2, c_out2;
  full_adder FA2(.a(a2), .b(b2), .cin(c_out1),
                 .sum_out(s_out2), .c_out(c_out2));

endmodule
```

Making a Module Instance

- Format:

```
<module name>  <instance name> ( <port connections> );
```

- Examples:
  - Connect by name


```
D_FF dff_0 (.q(out), .d(in), .clk(clk), .reset(R));
```

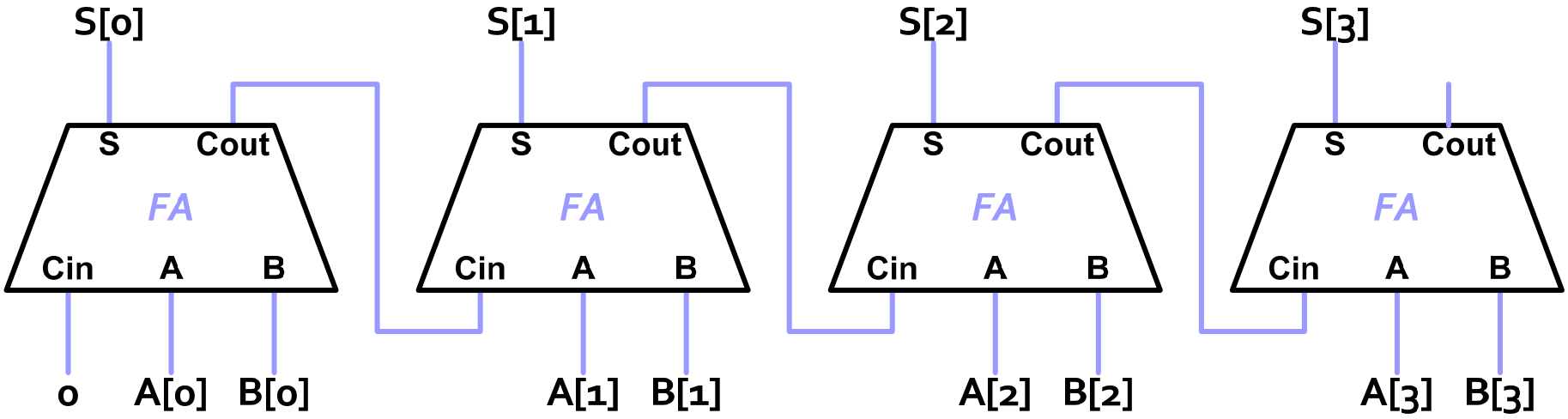
  - Connect by order

```
D_FF dff_0 (out, in, clk, R);
```

  - Note: we recommend to use "connect by name"

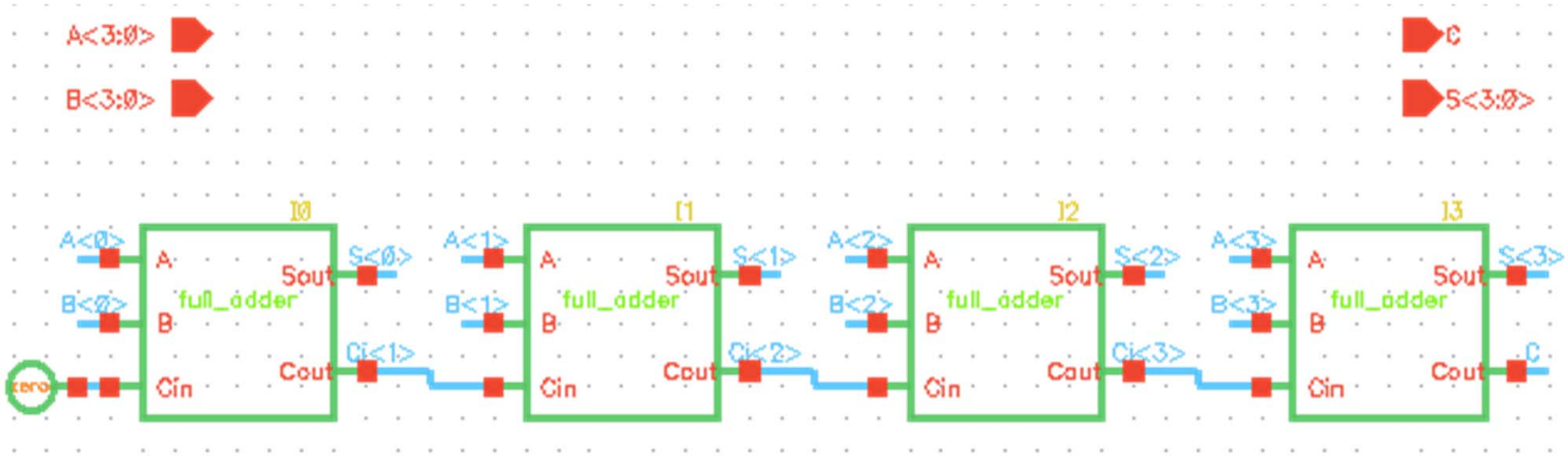
Exercise #6

- Build a 4-bit adder by combining the full adder modules you wrote in Exercise #1
  - Start with a skeleton in **models/add4_fa.sv**



Answer #6

- Located in **models/answer/add4_fa.sv**
- Verify its functionality by running **sim/tb_add4_fa**



Generate Modules

- When writing multiple identical modules, you can use ***generate*** block
  - Use ***genvar*** type for iterative module declaration

```
mod0 ss0(in[0], o[0]);
mod1 ss1(in[1], o[1]);
….
mod99 ss99(in[99], o[99]);
```

100 Lines!

```
genvar i;
generate
  for(i=0; i<100; i++) begin
       mod ss(in[i], o[i]);
  end
endgenerate
```

6 lines

Parameterized Modules

Parameter Definitions

Variable-width Signals using Parameters

```
module adder #(
    parameter in_width = 2,
    parameter out_width = 3
)(
    input[in_width-1:0] a,
    input[in_width-1:0] b,
    input cin,
    output[out_width-1:0] sum_out,
    output c_out
);
    assign {c_out, sum_out} = a+b+cin;
endmodule
```

Assignments: Blocking vs. Non-Blocking

- Inside an ***always*** block you can have
  - **Blocking assignment =**

    a group of them are evaluated sequentially
  - **Non-blocking assignment <=**

    are evaluated in parallel

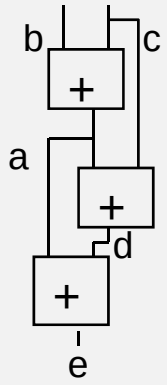
- Use non-blocking, since it's a better model for hardware!
  - Real gates operate independently
- But don't mix them within the same always block
  - It is a common source of confusion/errors

Good Practice with Assignments

- For combinational logic only use blocking assignments

```
always @*
   begin
      a = b + c; // evaluates first
      d = a + c;
      e = d + a; // evaluates last
   end
```

Blocking



- For sequential logic only use non-blocking assignments

```
always @(posedge Clk)
   begin
      a <= next_a;
      b <= a; // b receives a, not next_a
   end
```

Non-blocking