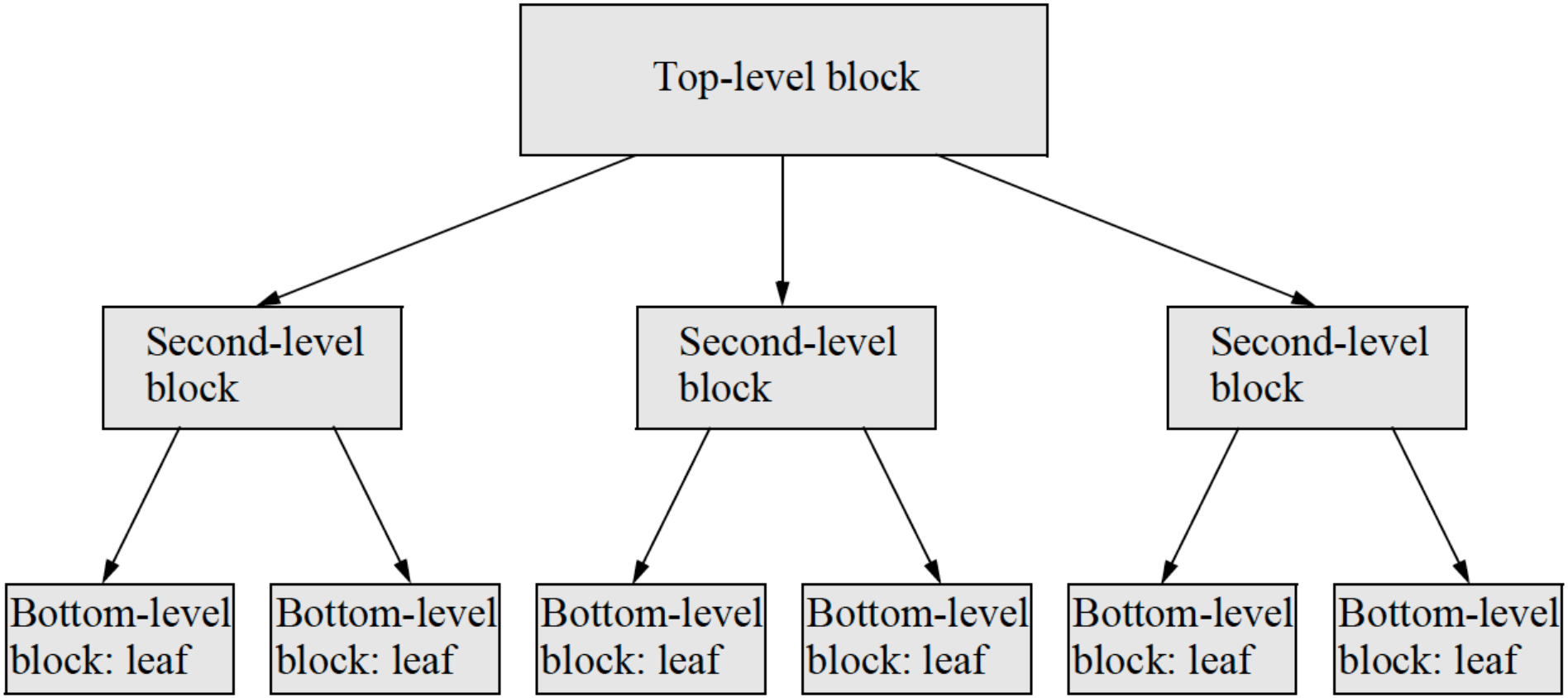
DIGITAL DESIGN WITH HDL

Chapter 1: Fundamentals

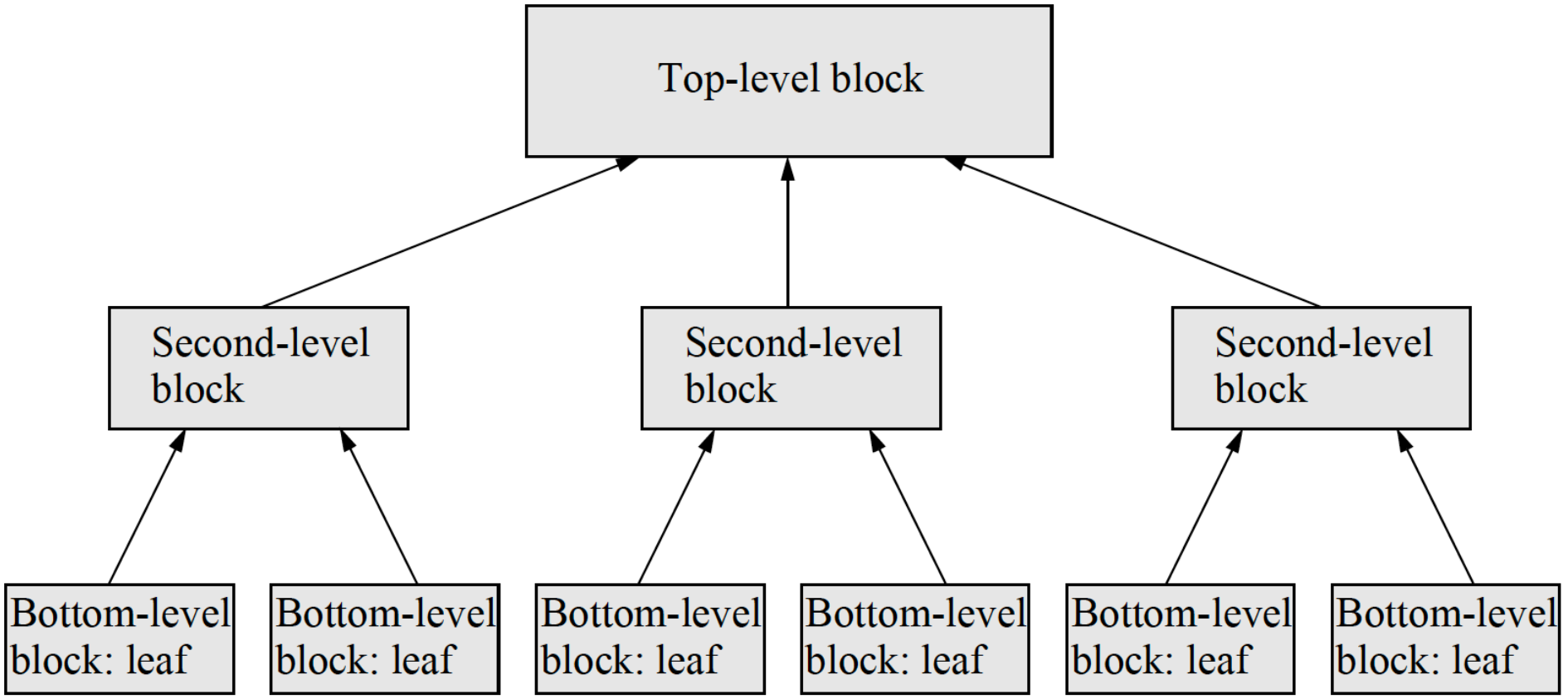
Design methodologies

- Two design methodologies:
  - Top-down design
    - Top-level blocked is identified
    - Next lower-level are defined
    - Bottom-level contains blocks cannot be further divided
  - Bottom-up design
    - The lowest level - leaf cells - is defined first
    - These blocks are used to build the next higher levels
    - Continue until the top-level block reached

Top-down design



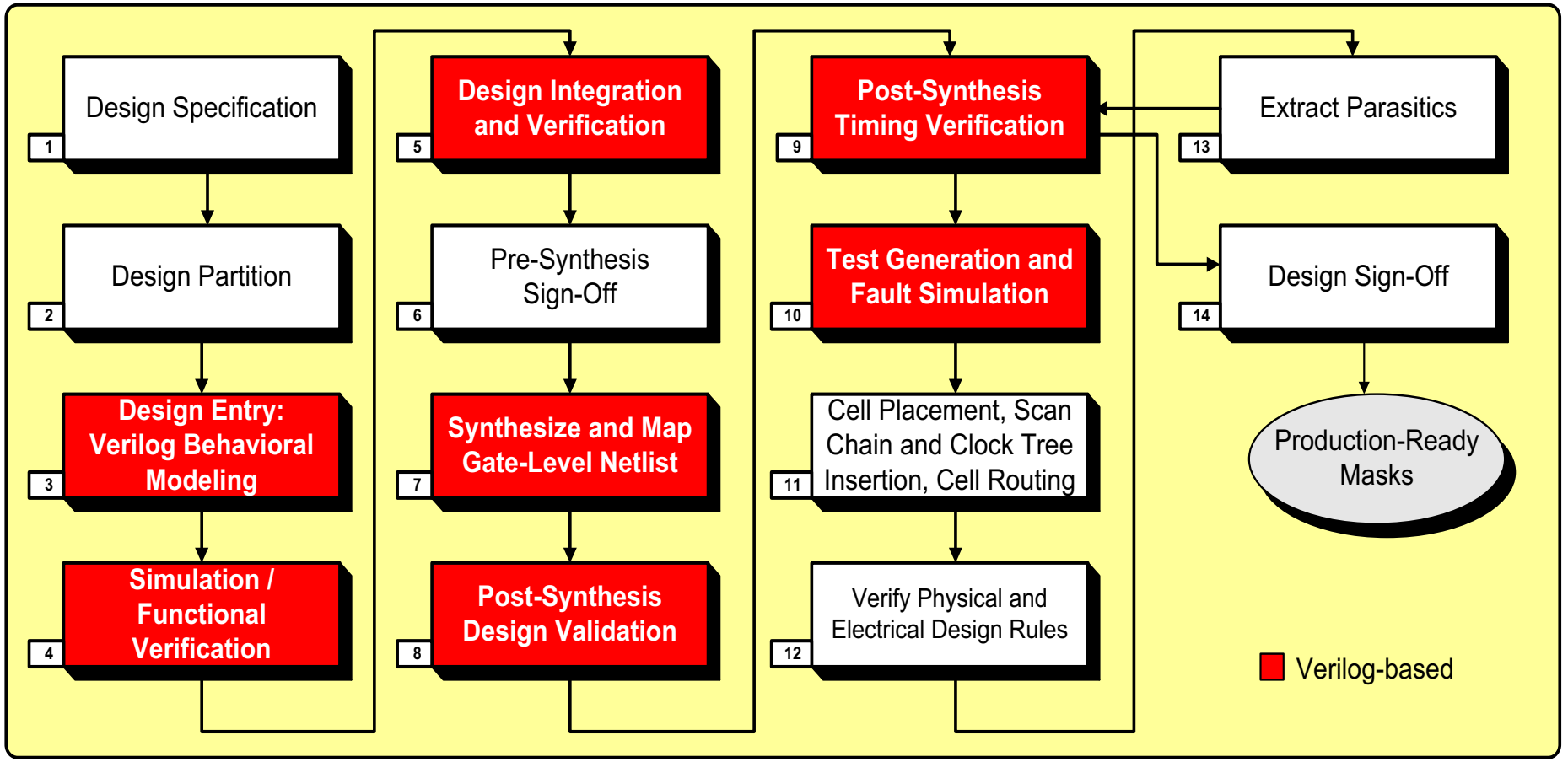
Bottom-up design



Overview of HDLs

- Hardware description languages (HDLs)
  - Are computer-based hardware description languages
  - Allow modeling and simulating the functional behavior and timing of digital hardware
  - Synthesis tools take an HDL description and generate a technology-specific netlist
- Two main HDLs used by industry
  - Verilog HDL (C-based, industry-driven)
  - VHSIC HDL or VHDL (Ada-based, defense/industry/ university-driven)

Why HDLs?

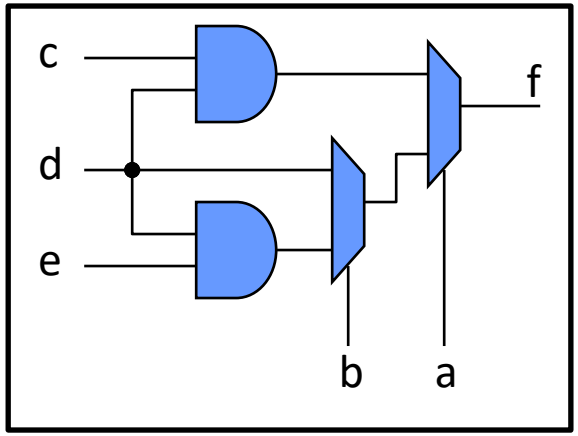

1
Design Specification
2
Design Partition
3
Design Entry: Verilog Behavioral Modeling
4
Simulation / Functional Verification
5
Design Integration and Verification
6
Pre-Synthesis Sign-Off
7
Synthesize and Map Gate-Level Netlist
8
Post-Synthesis Design Validation
9
Post-Synthesis Timing Verification
10
Test Generation and Fault Simulation
11
Cell Placement, Scan Chain and Clock Tree Insertion, Cell Routing
12
Verify Physical and Electrical Design Rules
13
Extract Parasitics
14
Design Sign-Off
Production-Ready Masks
Verilog-based

Why HDLs

- Takes a description of what a circuit does
- Creates the hardware to do it (semi-) automatically
  - During the synthesis process

```
if (a) f = c & d;
  else if (b) f = d;
  else f = d & e;
```

Synthesis process



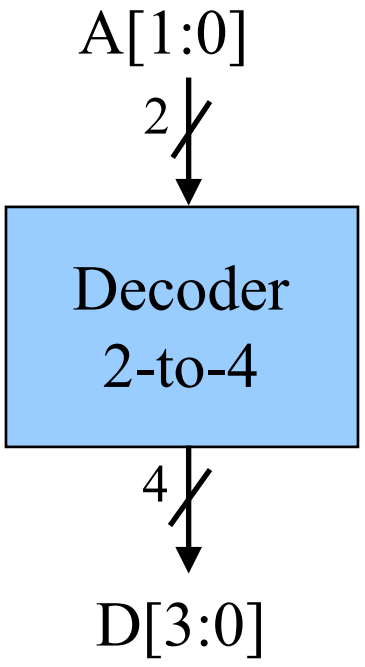
Verilog

- In this class, we will use the Verilog HDL
  - Used in academia and industry
  - Many principles we will discuss apply to any HDL
  - Once you can “think hardware”, you should be able to use any HDL fairly quickly
- Provides many descriptive styles
  - Structural
  - Register Transfer Level (RTL)
  - Behavioral
- C-like language
  - **NOT** programming language

Verilog module

- A circuit is a **module**
  - Could be simple or complex

```
module decoder_2_to_4 (A, D) ;

input [1:0] A ;
output [3:0] D ;

assign D = (A == 2'b00) ? 4'b0001:
           (A == 2'b01) ? 4'b0010:
           (A == 2'b10) ? 4'b0100:
           (A == 2'b11) ? 4'b1000;
endmodule
```



Module template

```
module <module name> (port list);
    declarations
        reg, wire, parameter,
        input, output, . . .
        . . .
    <module internals>
        statements
        initial, always, module instantiation, . . .
        . . .
endmodule
```

Verilog module example

Module name

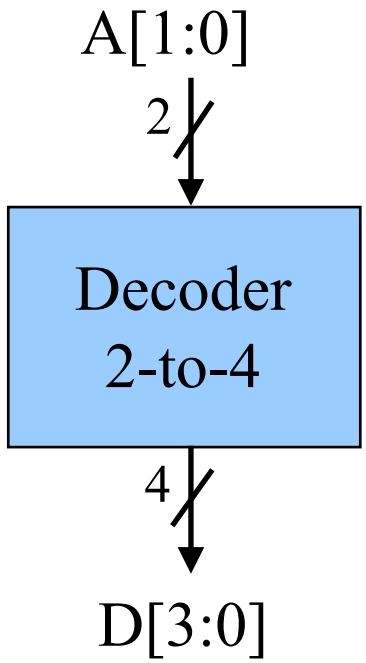
ports names of module

Port types

Port size

```
module decoder_2_to_4 (A, D) ;

input [1:0] A ;
output [3:0] D ;

assign D = (A == 2'b00) ? 4'b0001:
           (A == 2'b01) ? 4'b0010:
           (A == 2'b10) ? 4'b0100:
           (A == 2'b11) ? 4'b1000;

endmodule
```

Module content



Declaring a module

1. Start with the module keyword
2. Choose a descriptive module name (different from keywords)
3. Declare the ports (connectivity)
   - Choose descriptive signal names
   - Define types and size
4. Declare any internal signals
5. Write the content/functionality (the course)
6. Finalize by the endmodule keyword

Ports

- A signal is attached to every port
- Port types:
  - input: receiving arrival information
  - output: providing results
  - inout: functioning as both input or output (rarely used)
- Port size
  - Scalar: single bit (by default)
    - input cin;
  - Vector: multiple bits (bus), must specify
    - Syntax: [MSB:LSB]
    - Size: $|\text{MSB} - \text{LSB}| + 1$
    - Example:
      - output [3:0] sum;
      - input [1:4] a, b;

Module styles

- Modules (content/functionality) can be specified different ways
  - **Structural** – connect primitives and modules
  - **RTL** (Register transfer level) – use continuous assignments
  - **Behavioral** – use initial and always blocks
- A single module can use more than one method!

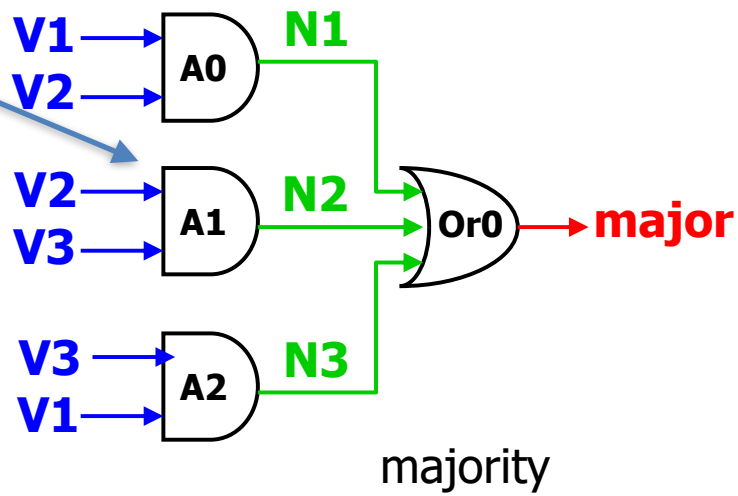
Structure model

- A schematic in text form
- Structural module can be designed using (**instances**):
  - **Primitive** gates (predefined simple gates)
  - User defined modules (previously designed and tested)
- Interconnections between instances are specified using nets data-type

Structural model example

```
module majority (major, V1, V2, V3);

   output major;
   input V1, V2, V3;
   wire N1, N2, N3;

   and A0 (N1, V1, V2),
       A1 (N2, V2, V3),
       A2 (N3, V3, V1);

   or  Or0  (major, N1, N2, N3);

endmodule
```

instances

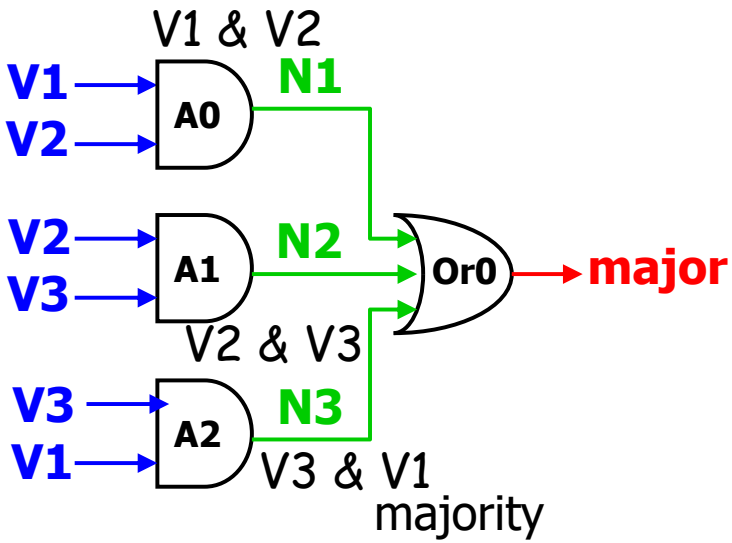


RTL model

- Data flow model
  - Design combinational logic only
  - Higher level of abstraction than the structural model
- Using the continuous assignment statement **assign**
  - Operators are used to make expressions

RTL model example

```
module majority (major, V1, V2, V3) ;

output major ;
input V1, V2, V3 ;

assign major = V1 & V2
             | V2 & V3
             | V1 & V3;
endmodule
```



Behavioral model

- Behavioral modeling is an abstraction of the functional operation of the design
  - Do not describe the implementation of the design
  - Outputs are characterized by their relationship to the inputs
  - Use initial and always statements
    - Procedural constructs

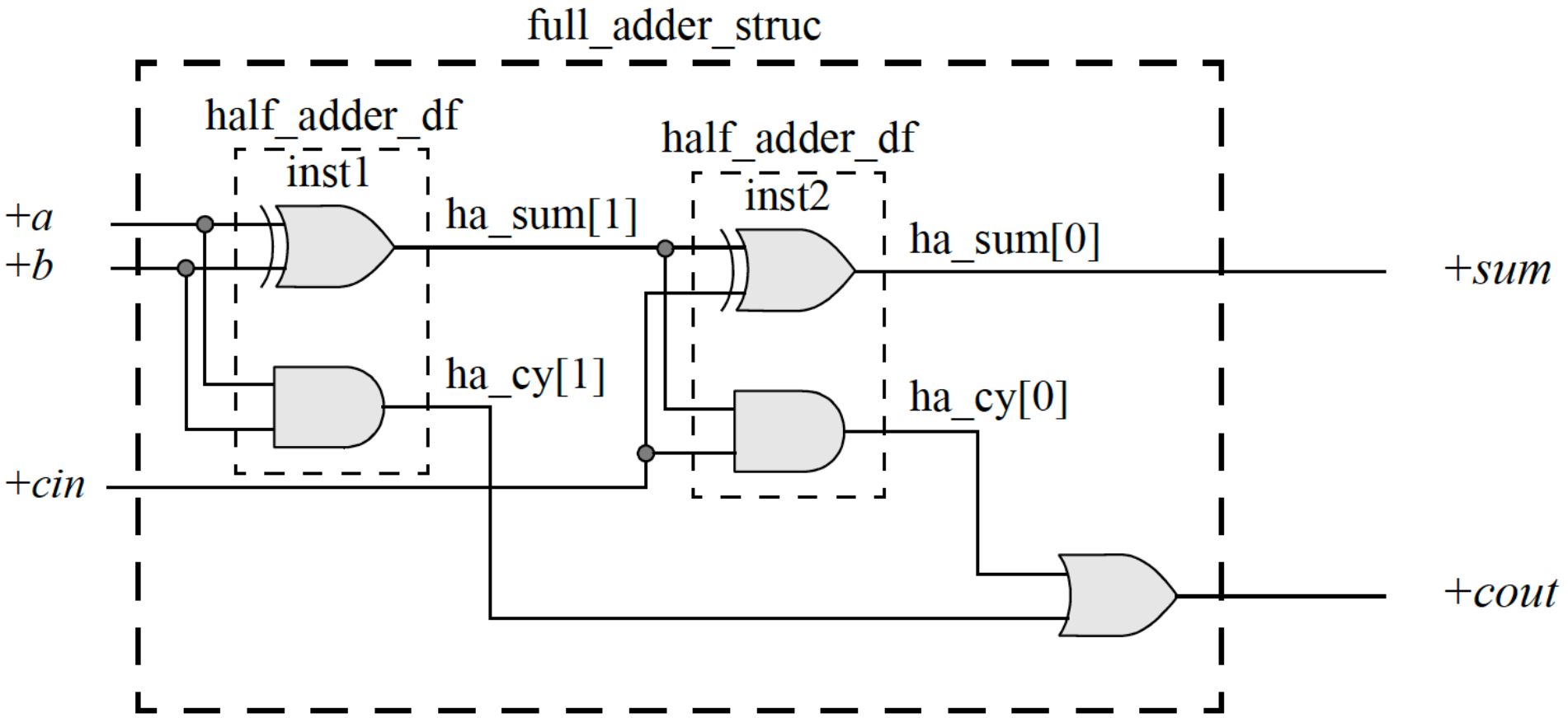
Behavioral model example

```
module majority (major, V1, V2, V3) ;

output reg major ;
input V1, V2, V3 ;

always @(V1, V2, V3) begin
  if (V1 && V2 || V2 && V3
        || V1 && V3) major = 1;
    else major = 0;
end

endmodule
```

New output type "reg" used to store values

Relationship between the output and their inputs

One more example



The full adder circuit

Structural model implementation

```
module fulladder_struc(a, b, cin, sum, cout);
    input a, b, cin;
    output sum, cout;
    wire ha_sum_1, ha_cy_1, ha_cy_0;

    xor(ha_sum_1, a, b);
    and(ha_cy_1, a, b);

    xor(sum, ha_sum_1, cin);
    and(ha_cy_0, ha_sum_1, cin);

    or(cout, ha_cy_1, ha_cy_0);

endmodule
```

half_adder_df sub-module

half_adder_df sub-module

RTL model implementation

```
module fulladder_rtl(a, b, cin, sum, cout);
    input a, b, cin;
    output sum, cout;

    assign sum = a ^  b ^ cin;
    assign cout = (a & b) | (a & cin) | (b & cin);

endmodule
```

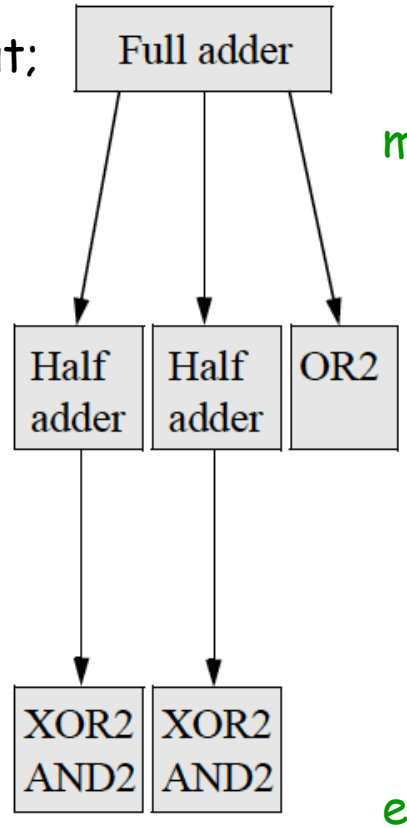
$$cout = (a \oplus b) \cdot cin + a \cdot b = (a \cdot b) \,|\, (a \cdot cin) \,|\, (b \cdot cin)$$

Behavioral model implementation

```
module fulladder_bhv(a, b, cin, sum, cout);
    input a, b, cin;
    output reg sum, cout;

    always @(a, b, cin) begin
        sum = a ^  b ^ cin;
        cout = (a & b) | (a & cin) | (b & cin);
    end

endmodule
```

Hierarchy design

```
module half_adder_struc(a, b, sum, cout);
    input a, b;
    output sum, cout;

    xor(sum, a, b);
    and(cout, a, b);

endmodule
```

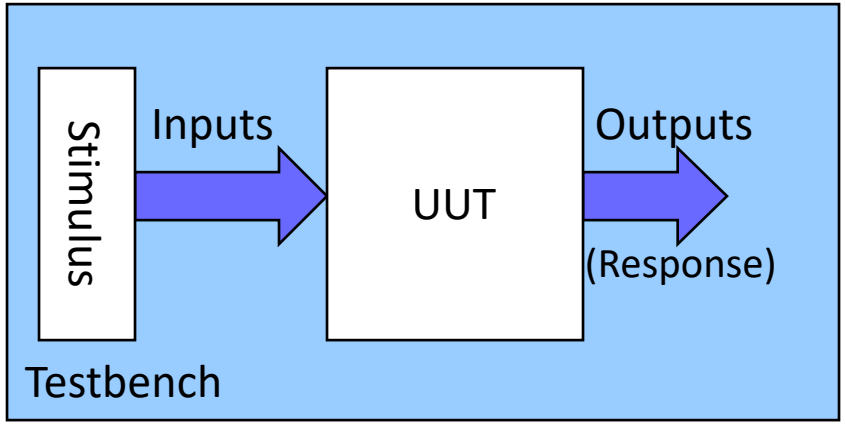


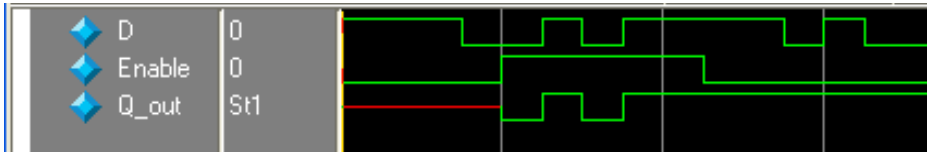

```
module fulladder_struc(a, b, cin, sum, cout);
    input a, b, cin;
    output sum, cout;
    wire c1, c2, s2;

    half_add_struc PARTSUM(a, b, s1, c1);
    half_add_struc SUM(s1, cin, sum, c2);

    or(cout, c2, c1);

endmodule
```

Instance name

Verification by simulation

- Testing a designed module to ensure its operations
  - Applying stimulus to inputs and checking outputs
- Use a test bench module
  - An instantiation of a unit under test
  - Verilog code to generate stimulus
  - Verilog code to monitor and display the response
    - Text
    - Waveform

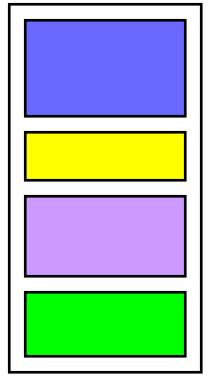


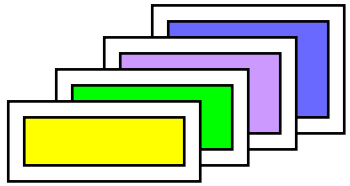



```
#0 x1=1'b0; x2=1'b0; x3=1'b0;
#10 x1=1'b0; x2=1'b0; x3=1'b1;
#10 x1=1'b0; x2=1'b1; x3=1'b0;
```

Hierarchical & source code

- Can have all modules in a single file
  - Module order doesn't matter!
  - Good for small designs
  - Not so good for bigger ones
  - Not so good for module reuse (cut & paste)
- Can break up modules into multiple files
  - Helps with organization
  - Lets you find a specific module easily
  - Great for module reuse (add file to project)





The end