

Digital Design with the Verilog HDL

Chapter 2: Introduction to Verilog

Binh Tran-Thanh

Department of Computer Engineering
Faculty of Computer Science and Engineering
Ho Chi Minh City University of Technology

January 15, 2024



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).



Synthesis of HDLs

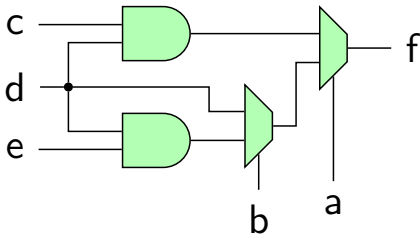
- Takes a description of what a circuit DOES
- Creates the hardware to DO it
- HDLs may LOOK like software, **but they're not!**
 - NOT a program
 - Doesn't "run" on anything
 - Though we do simulate them on computers
 - Don't confuse them!
- Also use HDLs to test the hardware you create
 - This is more like software



Describing Hardware!

- All hardware created during synthesis
 - Even if a is true, still computing d&e
- Learn to understand how descriptions translated to hardware

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



Why Use an HDL?

- More and more transistors can fit on a chip
 - Allows larger designs!
 - Work at transistor/gate level for large designs: hard
 - Many designs need to go to production quickly
- Abstract large hardware designs!
 - Describe what you need the hardware to do
 - Tools then design the hardware for you



Why Use an HDL?

- Simplified & faster design process
- Explore larger solution space
 - Smaller, faster, lower power
 - Throughput vs. latency
 - Examine more design tradeoffs
- Lessen the time spent debugging the design
 - Design errors still possible, but in fewer places
 - Generally easier to find and fix
- Can reuse design to target different technologies
 - Don't manually change all transistors for rule change



Other Important HDL Features

- Are highly portable (text)
- Are self-documenting (when commented well)
- Describe multiple levels of abstraction
- Represent parallelism
- Provides many descriptive styles
 - Structural
 - Register Transfer Level (RTL)
 - Behavioral
- Serve as input for synthesis tools



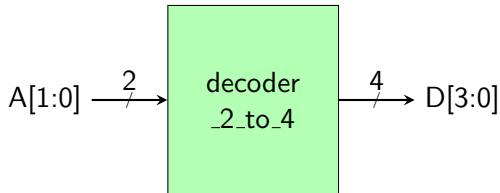
- In this class, we will use the Verilog HDL
 - Used in academia and industry
- VHDL is another common HDL
 - Also used by both academia and industry
- Many principles we will discuss apply to any HDL
- Once you can “think hardware”, you should be able to use any HDL fairly quickly



Verilog Module

In Verilog, a circuit is a **module**.

```
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
```



Declaring A Module

- Can't use keywords as module/port/signal names
 - Choose a descriptive module name
- Indicate the ports (connectivity)
- Declare the signals connected to the ports
 - Choose descriptive signal names
- Declare any internal signals
- Write the internals of the module (functionality)



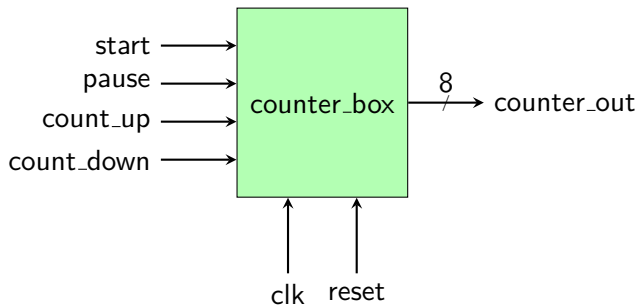
Declaring Ports

- A signal is attached to every port
- Declare type of port
 - **input**
 - **output**
 - **inout** (bidirectional)
- Scalar (single bit) - don't specify a size
 - **input** `cin`;
- Vector (multiple bits) - specify size using range
 - Range is MSB to LSB (left to right)
 - Don't have to include zero if you don't want to... (`D[2:1]`)
 - **output** `[7:0]` **OUT**;
 - **input** `[1:0]` **IN**;



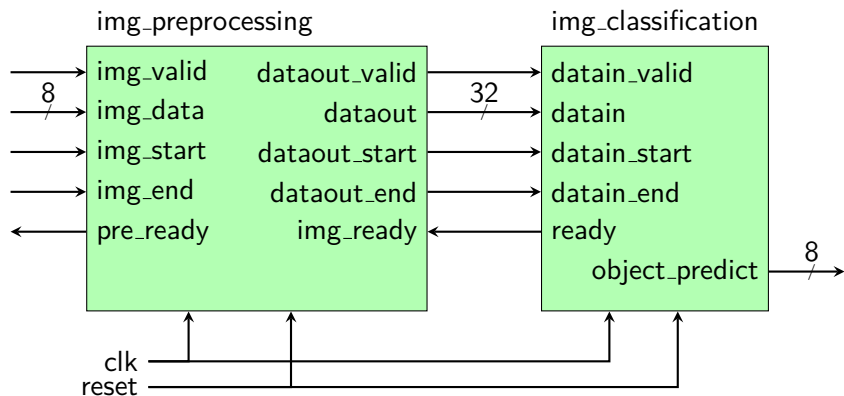
Your turn(1/2)

Using VerilogHDL to declare an interface (module name and ports) of following hardware



Your turn (2/2)

Using VerilogHDL to declare an interface (module name and ports) of following hardware



Module Styles

- Modules can be specified different ways
 - Structural – connect primitives and modules
 - RTL – use continuous assignments
 - Behavioral – use initial and always blocks
- A single module can use more than one method!
- What are the differences?



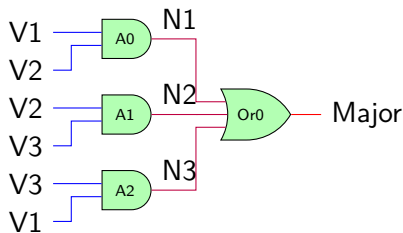
Structural

- A schematic in text form
- Build up a circuit from gates/flip-flops
 - Gates are primitives (part of the language)
 - Flip-flops themselves described behaviorally
- Structural design
 - Create module interface
 - Instantiate the gates in the circuit
 - Declare the internal wires needed to connect gates
 - Put the names of the wires in the correct port locations of the gates
 - For primitives, outputs always come first



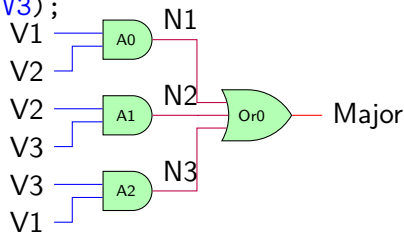
Structural Example

```
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
```



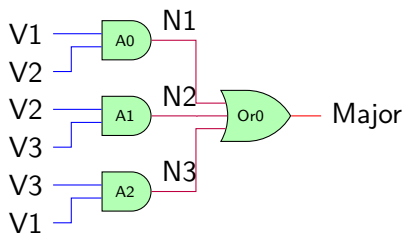
RTL Example

```
module majority (major, V1, V2, V3);  
  output major;  
  input V1, V2, V3;  
  
  assign major = V1 & V2  
                | V2 & V3  
                | V1 & V3;  
  
endmodule
```

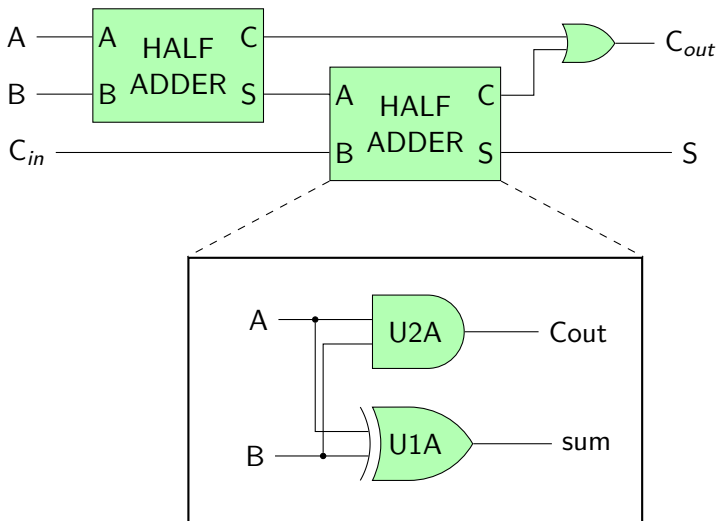


Behavioral Example

```
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)) begin  
            major = 1;  
        end  
        else begin  
            major = 0;  
        end  
    end  
endmodule
```



Adder Example



Full Adder: Structural

```
module half_add (X, Y,  
    S, C);  
    input X, Y;  
    output S, C;  
    xor SUM (S, X, Y);  
    and CARRY (C, X, Y);  
endmodule
```

```
module full_add (A, B, CI, S, CO);  
    input A, B, CI;  
    output S, CO;  
    wire S1, C1, C2;  
  
    // full adder from 2 half-adders  
    half_add PARTSUM (A, B, S1, C1);  
    half_add SUM (S1, CI, S, C2);  
  
    // OR gate for the carry  
    or CARRY (CO, C2, C1);  
endmodule
```



Full Adder: RTL/Dataflow

```
module fa_rtl (A, B, CI, S, CO);  
  input A, B, CI;  
  output S, CO;  
  
  // use continuous assignments  
  assign S = A ^ B ^ CI;  
  assign CO = (A & B) | (A & CI) | (B & CI);  
endmodule
```



Full Adder: Behavioral

- Circuit “reacts” to given events (for simulation)
 - Actually list of signal changes that affect output

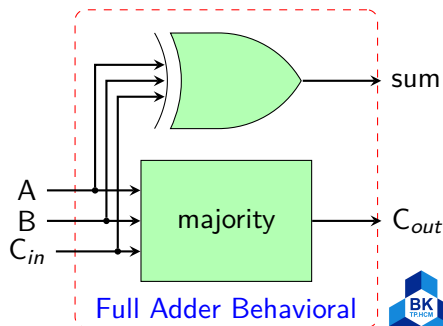
```
module fa_bhv (A, B, CI, S, CO);  
  input A, B, CI;  
  output S, CO;  
  reg S, CO; // explained in later lecture - "holds" values  
  
  // use procedural assignments  
  always @(A or B or CI) begin  
    S = A ^ B ^ CI;  
    CO = (A & B) | (A & CI) | (B & CI);  
  end  
endmodule
```



Full Adder: Behavioral

- IN SIMULATION
 - When **A**, **B**, or **C** change, **S** and **CO** are recalculated
- IN REALITY
 - Combinational logic – no “waiting” for the trigger
 - **Constantly** computing - think transistors and gates!
 - Same hardware created for this and RTL example

```
always @ (A or B or CI)
begin
    S = A ^ B ^ CI;
    CO = (A & B) | (A & CI)
        | (B & CI);
end
```



Structural Basics: Primitives

- Build design up from the gate/flip-flop/latch level
 - Flip-flops actually constructed using Behavioral
- Verilog provides a set of gate primitives
 - and, nand, or, nor, xor, xnor, not, buf, bufif1, etc.
 - Combinational building blocks for structural design
 - Known “behavior”
 - Cannot access “inside” description
- Can also model at the transistor level
 - Most people don't, we won't



Primitives

- No declarations - can only be instantiated
- Output port appears before input ports
- Optionally specify: instance name and/or delay (discuss delay later)

```
and N25 (Z, A, B, C); // name specified
```

```
and #10 (Z, A, B, X),  
        (X, C, D, E); // delay specified, 2 gates
```

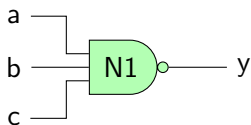
```
and #10 N30 (Z, A, B); // name and delay specified
```



Verilog Primitives

- 26 pre-defined primitives
- Output is the **first port**

n-input	n-output 3-states
and	buf
nand	not
or	bufif0
nor	bufif1
xor	notif0
xnor	notif1



Usage:

```
nand (y, a, b, c);  
nand N1(y, a, b, c);
```

- Keyword: **nand**
- Output: **y**
- Input: **a, b, c**
- Ending mark: **;**
- Instance name (optional): **N1**



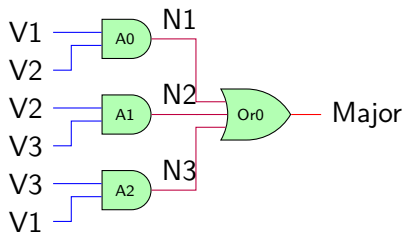
Syntax For Structural Verilog

- First declare the interface to the module
 - Module keyword, module name
 - Port names/types/sizes
- Next, declare any internal wires using “wire”
 - `wire [3:0] partialsum;`
- Then **instantiate** the primitives/submodules
 - Indicate which signal is on which port



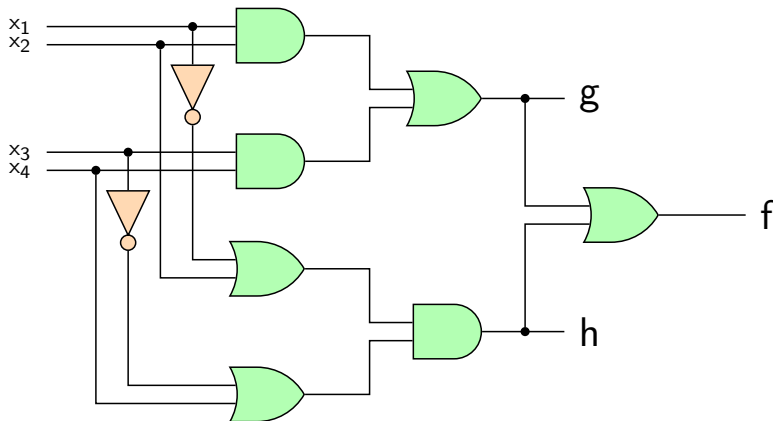
Again: Structural Example

```
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
```



Your turn

Using the Verilog structural style, describe the following circuit



Example: Combinational Gray code

Using the Verilog structural style, describe the following expressions

$$S_2^+ = \overline{Rst} \cdot S_2 \cdot S_0 + \overline{Rst} \cdot S_1 \cdot \overline{S_0}$$

$$S_1^+ = \overline{Rst} \cdot \overline{S_2} \cdot S_0 + \overline{Rst} \cdot S_1 \cdot \overline{S_0}$$

$$S_0^+ = \overline{Rst} \cdot \overline{S_2} \cdot \overline{S_1} + \overline{Rst} \cdot S_2 \cdot S_1$$



Datatypes

- Two categories
 - Nets
 - “Registers”
- Only dealing with nets in structural Verilog
- “Register” datatype doesn't actually imply an actual register...
 - Will discuss this when we discuss Behavioral Verilog



Net Types

- **wire**: most common, establishes connections
 - Default value for all signals
- **tri**: indicates will be output of a tri-state
 - Basically same as “wire”
- **supply0**, **supply1**: ground & power connections
 - Can imply this by saying “0” or “1” instead
 - `xor xorgate(out, a, 1'b1);`
- **wand**, **wor**, **triand**, **trior**, **tri0**, **tri1**, **triereg**
 - Perform some signal resolution or logical operation
 - Not used in this course

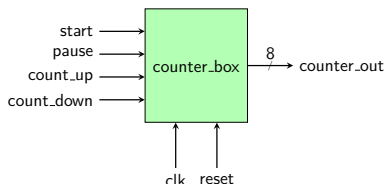


Structural Verilog: Connections

- "Positional" or "Implicit" port connections
 - Used for primitives (first port is output, others inputs)
 - Can be okay in some situations
- Designs with very few ports
- Interchangeable input ports (and/or/xor gate inputs)
 - Gets confusing for large #s of ports
- Can specify the connecting ports by name
 - Helps avoid "misconnections"
 - Don't have to remember port order
 - Can be easier to read
 - `.< port name>(<signal name>)`



Connections Examples



```
module counter_box( input clk, reset, start, pause,
                    input counter_up, counter_down,
                    output [7:0] counter_out);
```

- **Variables** defined in upper level module

```
module top_module_name (.....);
    wire clk, rst;
    wire start, pause, counter_up, counter_down;
    wire [7:0] counter_out;
    .....
endmodule
```



Connections Examples

```
module counter_box( input clk, reset, start, pause,
                    input counter_up, counter_down,
                    output [7:0] counter_out);
```

- Connect module **by position**.

```
module top_module_name (.....);
    wire clk, rst;
    wire start, pause, counter_up, counter_down;
    wire [7:0] counter_out;

    counter_box U1(clk, rst, start, pause,
                  counter_up, counter_down,
                  counter_out);

endmodule
```



Connections Examples

```
module counter_box( input clk, reset, start, pause,
                    input counter_up, counter_down,
                    output [7:0] counter_out);
```

- Connect module **by name**.

```
module top_module_name (.....);
    wire clk, rst;
    wire start, pause, counter_up, counter_down;
    wire [7:0] counter_out;

    counter_box U2(.clk(clk), .reset(rst), .pause(pause),
                  .counter_up(counter_up), .counter_down(counter_down),
                  .counter_out(counter_out), .start(start));

endmodule
```

Both cases are the same connections



Empty Port Connections(1/2)

```
module top_module_name_emply_example (.....);  
    wire clk, rst;  
    wire start, pause, counter_up, counter_down;  
    wire [7:0] counter_outU1, counter_outU2,;  
    // missing input connections,  
    // reset, pause, and counter_down are high impedance (z)  
    counter_box U3(.clk(clk), .reset(),  
        .counter_up(counter_up), .counter_down(),  
        .counter_out(counter_outU1), .start(start));  
  
    // missing output connections,  
    // counter out [7:5] unused  
    counter_box U4(.clk(clk), .reset(), .pause(pause),  
        .counter_up(counter_up), .counter_down(counter_down),  
        .counter_out(counter_outU2[4:0]), .start(start));  
  
endmodule
```



Empty Port Connections(2/2)

- General rules
 - Empty input ports \Rightarrow high impedance state (z)
 - Empty output ports \Rightarrow output not used
- Specify all input ports anyway!
 - Usually don't want z as input
 - Clearer to understand & find problems
- Helps if no connection to name port, but leave empty:

```
counter_box U4(.clk(clk), .reset(1'b0), .pause(pause),  
  .counter_up(counter_up), .counter_down(counter_down),  
  .counter_out(counter_outU2), .start(start));
```

```
counter_box U5(.clk(clk), .reset(), .pause(pause),  
  .counter_up(counter_up), .counter_down(counter_down),  
  .counter_out(counter_outU2), .start(start));
```



Hierarchy

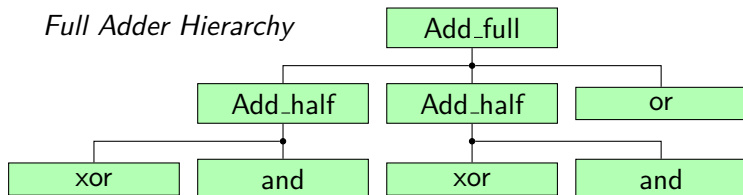
- Any Verilog design you do will be a module
- This includes testbenches!
- Interface (“black box” representation)
 - Module name, ports
- Definition
 - Describe functionality of the block
 - Includes interface
- Instantiation
 - Use the module inside another module



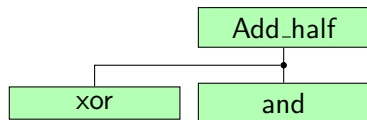
Hierarchy

- Build up a module from smaller pieces
 - Primitives
 - Other modules (which may contain other modules)
- Design: typically top-down
- Verification: typically bottom-up

Full Adder Hierarchy



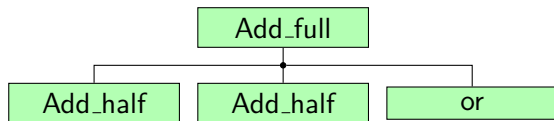
Add_half Module



```
module Add_half(c_out, sum, a, b);  
    output sum, c_out;  
    input a, b;  
  
    xor sum_bit(sum, a, b);  
    and carry_bit(c_out, a, b);  
  
endmodule
```



Add_full Module



```
module Add_full(c_out, sum, a, b, c_in);  
    output sum, c_out;  
    input a, b, c_in;  
    wire w1, w2, w3;  
  
    Add_half AH1(.sum(w1), .c_out(w2), .a(a), .b(b));  
    Add_half AH2(.sum(sum), .c_out(w3), .a(c_in), .b(w1));  
    or carry_bit(c_out, w2, w3);  
  
endmodule
```



Can Mix Styles In Hierarchy!

```
module Add_half_bhv(c_out,  
    sum, a, b);  
    output reg sum, c_out;  
    input a, b;  
  
    always @(a, b)  
    begin  
        sum = a ^ b;  
        c_out = a & b;  
    end  
endmodule
```

```
module Add_full_mix(c_out, sum,  
    a, b, c_in);  
    output sum, c_out;  
    input a, b, c_in;  
    wire w1, w2, w3;  
  
    Add_half_bhv AH1(.sum(w1),  
        .c_out(w2), .a(a), .b(b));  
    Add_half_bhv AH2(.sum(sum),  
        .c_out(w3), .a(c_in), .b(w1));  
    assign c_out = w2 | w3;  
endmodule
```



Hierarchy And Scope

- Parent cannot access “internal” signals of child
- If you need a signal, must make a port!

Example:

Detecting overflow

Overflow = cout

XOR cout6

Must output

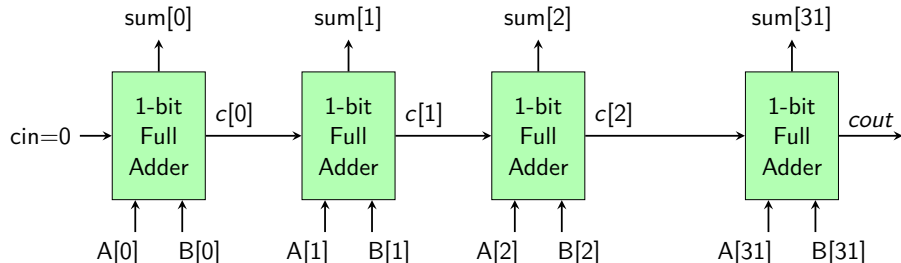
overflow or cout6!

```
module add8bit(cout, sum, a, b);  
    output [7:0] sum;  
    output cout;  
    input [7:0] a, b;  
    wire cout0, cout1, ..., cout6;  
    FA A0(cout0, sum[0], a[0], b[0], 1'b0);  
    FA A1(cout1, sum[1], a[1], b[1], cout0);  
    ...  
    FA A7(cout, sum[7], a[7], b[7], cout6);  
endmodule
```



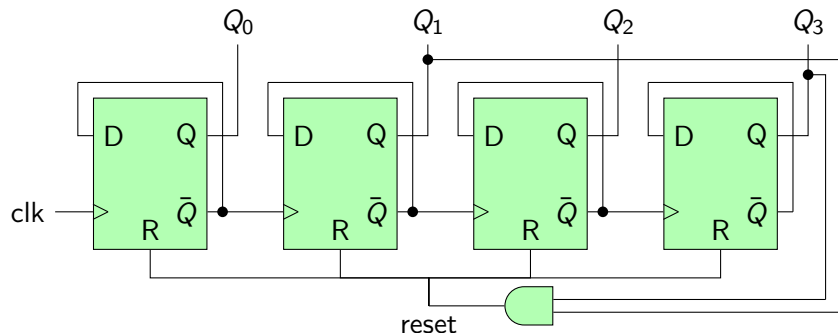
Homework (1)

Using `module Add_half`, `structural` styles to describe a 32-bit adder.



Homework (2)

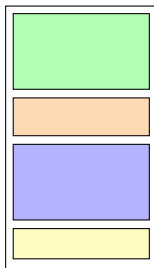
Using `D_FF(D, clk, async_reset, Q)`, **structural** styles to describe a following BCD counter.



Hierarchy And 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)

