Verilog Tutorial

Hardware Description Languages

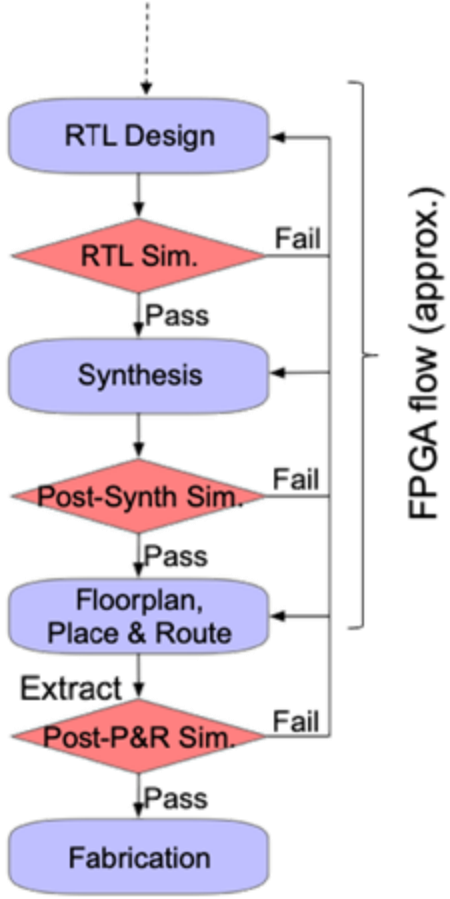
- Verilog:
  - Simple C-like syntax for structural and behavior hardware constructs
  - Mature set of commercial tools for synthesis and simulation
  - **Used in this course!**
- VHDL:
  - Semantically very close to Verilog
  - More syntactic overhead
  - Extensive type system for "synthesis time" checking
- System Verilog:
  - Enhances Verilog with strong typing along with other additions

Verilog: Brief History

- Developed by Gateway in 1985, acquired by Cadence in 1989.
- Originally for simulation; synthesis support came later.
- Released publicly in 1990 to compete with VHDL (developed by the US Department of Defense).
- Became popular in Silicon Valley due to strong tool support and C-like syntax.
- VHDL is still popular within the government, in Europe and Japan, and some universities.
- Most major CAD frameworks now support both.

Logic Synthesis

- Verilog and VHDL started out as simulation languages but soon programs were written to automatically convert Verilog code into low-level circuit descriptions (netlists).
- Synthesis converts Verilog (or other HDL) descriptions to an implementation using technology-specific primitives:
  - For FPGA: LUTs, FlipFlops, and BRAMs.
  - For ASICs: standard cells and memory macros.



Agenda

- Combinational Circuits
  - Assign statement
  - Always blocks
    - Case statement
  - Generator
- Sequential Circuits
  - Latches and FlipFlops
  - Always block
    - Sensitivity list
  - Blocking vs Nonblocking

Verilog Introduction

- Modules define components in a circuit.
- Two ways to describe modules:
  - Structural Verilog:
    - Defines sub-components and connections (text-based schematic).
    - Precise control but tedious to write and read.
    - Needed for FPGA/ASIC mapping.
  - Behavioral Verilog:
    - Describes what a component does, not how.
    - Easier to write but depends on synthesis tools.
- Modular Design:
  - Hierarchical structure with a top-level module as the full design or testbench.

Combinational Logic

- Output is a function of only the current inputs.
- No memory or state involved.
- The same inputs always produce the same outputs.
- No clock required—outputs update immediately with input changes.

Verilog Modules and Instantiation

- Modules define circuit components.
- Instantiation defines hierarchy of the design.

name

port list

keywords

port declarations (input, output, or inout)

Instance of addr_cell

```
module addr_cell (a, b, cin, s, cout);
  input  a, b, cin;
  output s, cout;
endmodule

module adder (A, B, S);
  addr_cell ac1 ( ... connections ... );
endmodule
```

```
module addr_cell(
    input  a,
    input  b,
    input  cin,
    output s,
    output cout
);
endmodule
```

You can declare the ports with their directions and types directly in the port list. **Recommended!**

Note: A module is not a function in the C sense. There is no call and return mechanism. Think of it more like a hierarchical data structure.

Structural Model – 1-bit Adder Example

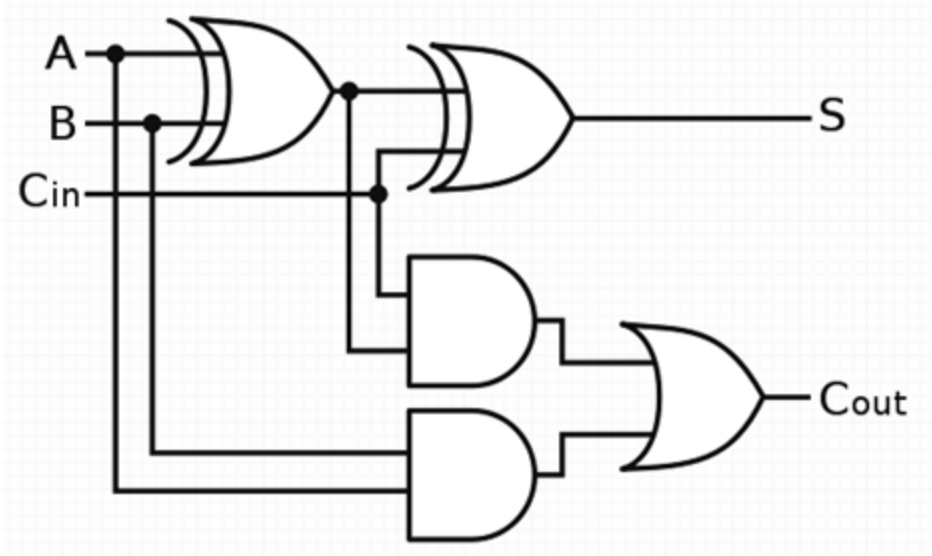
```
module addr_cell(
    input  a,
    input  b,
    input  cin,
    output s,
    output cout
);

    wire w1, w2, w3;

    xor g1 (w1, a, b);
    xor g2 (s, w1, cin);
                                Build-in gates!
    and g3 (w2, a, b);
    and g4 (w3, w1, cin);

    or g5 (cout, w2, w3);

endmodule
```



Notes:

- The instantiated gates are not "executed". They are active always.
- Undeclared variables assumed to be wires. Don't let this happen to you!

Simple Behavioral Model

```
module foo (
    input in1,
    input in2,
    output out
);

    assign out = in1 & in2;

endmodule
```

"continuous assignment"
Connects out to be the logical "and" of in1 and in2.

- Short-hand for explicit instantiation of bit-wise"and" gate (in this case).
- The assignment continuously happens, therefore any change on the rhs is reflected in out immediately
- Not like an assignment in C that takes place when the program counter gets to that place in the program.

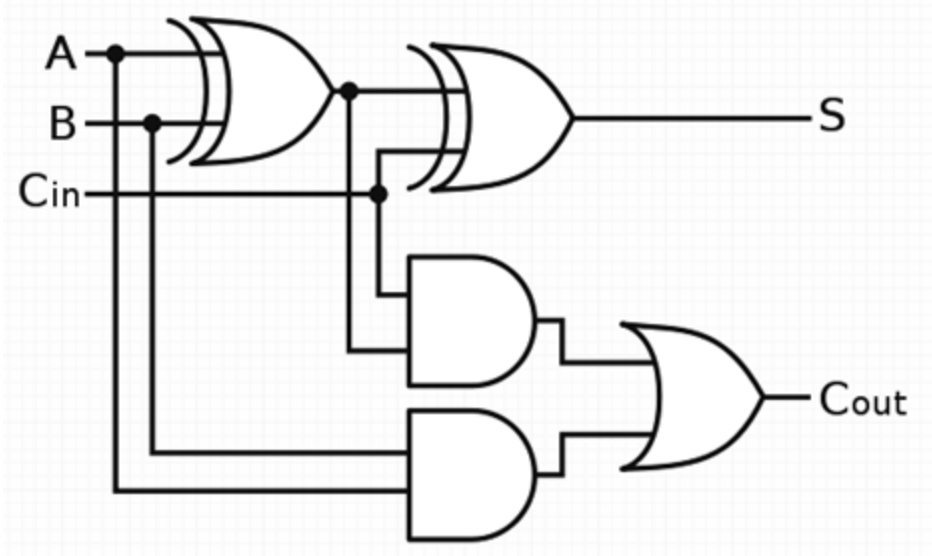
Behavioral Model – 1-bit Adder Example

```
module addr_cell(
    input  a,
    input  b,
    input  cin,
    output s,
    output cout
);

    wire w1, w2, w3;

    assign w1 = a ^ b;
    assign s = w1 ^ cin;
    assign w2 = a & b;
    assign w3 = w1 & cin;
    assign cout = w2 | w3;

endmodule
```



Even Simpler!

```
module addr_cell(
    input  a,
    input  b,
    input  cin,
    output s,
    output cout
);

    assign {cout, s} = a + b + cin;

endmodule
```

Instantiation, Signal Array, Named ports

- Example: 4-bit Ripple-Carry Adder.
  - S[4:0] = A + B (5-bit sum)

```
module adder4 (
    input  [3:0] A,
    input  [3:0] B,
    output [4:0] S
);

    wire c1, c2, c3;

    addr_cell ac0 (.a(A[0]), .b(B[0]), .cin(1'b0),  .s(S[0]), .cout(c1));
    addr_cell ac1 (.a(A[1]), .b(B[1]), .cin(c1),    .s(S[1]), .cout(c2));
    addr_cell ac2 (.a(A[2]), .b(B[2]), .cin(c2),    .s(S[2]), .cout(c3));
    addr_cell ac3 (.a(A[3]), .b(B[3]), .cin(c3),    .s(S[3]), .cout(S[4]));

endmodule
```

Signal array. Declares A[3], A[2], A[1], A[0]

Named ports. Highly recommended. → prevent errors if port order changes.

Verilog Operators

Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
!	logical negation	logical
~	negation	bit-wise
&	reduction AND	reduction
|	reduction OR	reduction
~&	reduction NAND	reduction
~|	reduction NOR	reduction
^	reduction XOR	reduction
~^ or ^~	reduction XNOR	reduction
+	unary (sign) plus	arithmetic
-	unary (sign) minus	arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
*	multiply	arithmetic
/	divide	arithmetic
%	modulus	arithmetic
+	binary plus	arithmetic
-	binary minus	arithmetic
<<	shift left	shift
>>	shift right	shift
>	greater than	relational
>=	greater than or equal to	relational
<	less than	relational
<=	less than or equal to	relational
==	case equality	equality
!=	case inequality	equality
&	bit-wise AND	bit-wise
^	bit-wise XOR	bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

Continuous Assignment Examples

- Assign values whenever there is a change in the RHS.
- Model combinational logic without specifying an interconnection of gates.

```
wire [3:0] A, X, Y, R, Z;
wire [7:0] P;
wire r, a, cout, cin;
```

```
assign R = X | (Y & ~Z);
assign r = &X;
assign R = (a == 1'b0) ? X : Y;
assign P = 8'hff;
assign P = X * Y;
assign P[7:0] = {4{X[3]}, X[3:0]};
assign {cout, R} = X + Y + cin;
assign Y = A << 2;
assign Y = {A[1], A[0], 1'b0, 1'b0};
```

Non-Continuous Assignments

- A bit unusual from a hardware specification point of view.
- Shows off Verilog roots as a simulation language.

"reg" type declaration. Not really a register in this case. Just a Verilog idiosyncrasy.

brackets multiple statements (not necessary in this example.

```
module and_or_gate (
    input  in1,
    input  in2,
    input  in3,
    output reg out
);

    always @(in1 or in2 or in3) begin
        out = (in1 & in2) | in3;
    end

endmodule
```

Sensitivity list, triggers the action in the body

Styles of doing combo logic

Assign 'Explicit Style'

```
wire   empty;

assign empty = (cnt==0) && !wr_val;
```

```
reg    [ST_W-1:0]   state;
wire   [ST_W-1:0].  state_nxt;

assign state_nxt = val_1_in ? ACCEPT_ST:
                   val_2_in ? WAIT_ST  :
                              state ;
```

Always @(*) 'Serial Style'

```
reg   empty;

always @(*) begin
  empty = (cnt==0) && !wr_val;
end
```

```
reg [ST_W-1:0] state, state_next;

always @(*) begin
  state_next = state; //default
  if (val_1_in)
    state_next = ACCEPT_ST;
  else if (val_2_in)
    state_next = WAIT_ST;
end
```

Always Blocks

- Always blocks give us some constructs that are impossible or awkward in continuous assignments.

```
module mux4 (
    input  in0,
    input  in1,
    input  in2,
    input  in3,
    input  [1:0] select,
    output reg out
);
    always @ (in0 or in1 or in2 or in3 or select) begin
        case (select)
            2'b00: out = in0;
            2'b01: out = in1;
            2'b10: out = in2;
            2'b11: out = in3;
        endcase
    end
endmodule // mux4
```

The statement(s) corresponding to whichever constant matches "select" get applied.

Couldn't we just do this with nested "if"s? Well yes and no!

Always Blocks

Nested if structure leads to "priority logic" structure:
with different delays for different inputs
(in3 to out delay > than in0 to out delay).
Case version treats all inputs the same.

- Nested if-else example:

```
module mux4 (
    input  in0,
    input  in1,
    input  in2,
    input  in3,
    input  [1:0] select,
    output reg out
);
    always @ (in0 or in1 or in2 or in3 or select) begin
        if (select == 2'b00)
            out = in0;
        else if (select == 2'b01)
            out = in1;
        else if (select == 2'b10)
            out = in2;
        else
            out = in3;
    end

endmodule // mux4
```

Generate

```
adder #(.N(64)) adder64 ( ... );
```
Overwrite parameter N at instantiation.

```
module adder (
    parameter N = 4;
    input  [N-1:0] A,
    input  [N-1:0] B,
    output [N:0] S
);

    wire [N:0] carry;

    genvar i;
    generate
        for (i = 0; i < N; i = i + 1) begin : bit
            addr_cell ac (.a(A[i]), .b(B[i]), .cin(carry[i]), .s(S[i]), .cout(carry[i+1]));
        end
    endgenerate

    assign carry[0] = 1'b0;
    assign S[N] = carry[N];

endmodule
```

Declare a parameter with default value.
Note: this is not a port. Acts like a "synthesis-time" constant.

Replace all occurrences of "4" with "N".

variable exists only in the specification - not in the final circuit.

For-loop creates instances (with unique names)

Keyword that denotes synthesis-time operations

- **This is NOT a sequential for loop, this is just to replicate hardware! All in parallel!**
- **You can declare signals inside this scope**

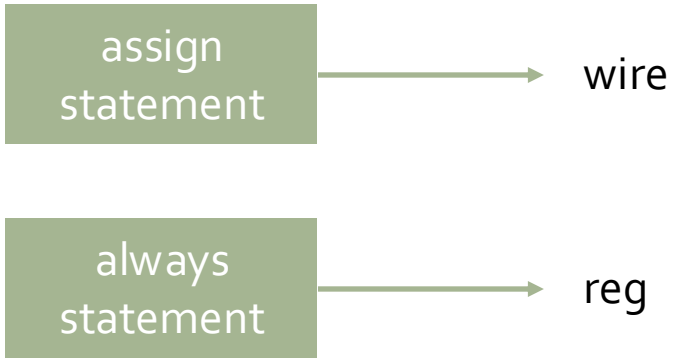
Simplified Verilog Guidelines

- Combinational logic:
  - Continuous Assignment:

    ```
    assign a = b & c;
    ```

  - Always block with @(*)

    ```
    always @(*) begin
            a = b & c; // blocking statement
    end
    ```



Are these Combinational Circuits Correct?

```
always @(a or b) begin
    out = a + b + c;
end
```

```
assign out = in & out;
```

```
always @(*) begin
    case (addr[1:0]):
        2'd0: out = a;
        2'd1: out = b;
        2'd2: out = c;
    endcase
end
```

Inside always blocks

Case statements

BAD!!!

```
case (addr[1:0]):
    2'd0: out = a;
    2'd1: out = b;
    2'd2: out = c;
endcase
```

GOOD!

```
case (addr[1:0]):
    2'd0: out = a;
    2'd1: out = b;
    2'd2: out = c;
    2'd3: out = d;
endcase
```

Important notes:

- Used in **always** blocks
- **A latch will be generated if *case* is not full and there is no default statement**

GOOD TOO!

```
case (addr[1:0]):
    2'd0: out = a;
    2'd1: out = b;
    2'd2: out = c;
    default: out = d;
endcase
```

Agenda

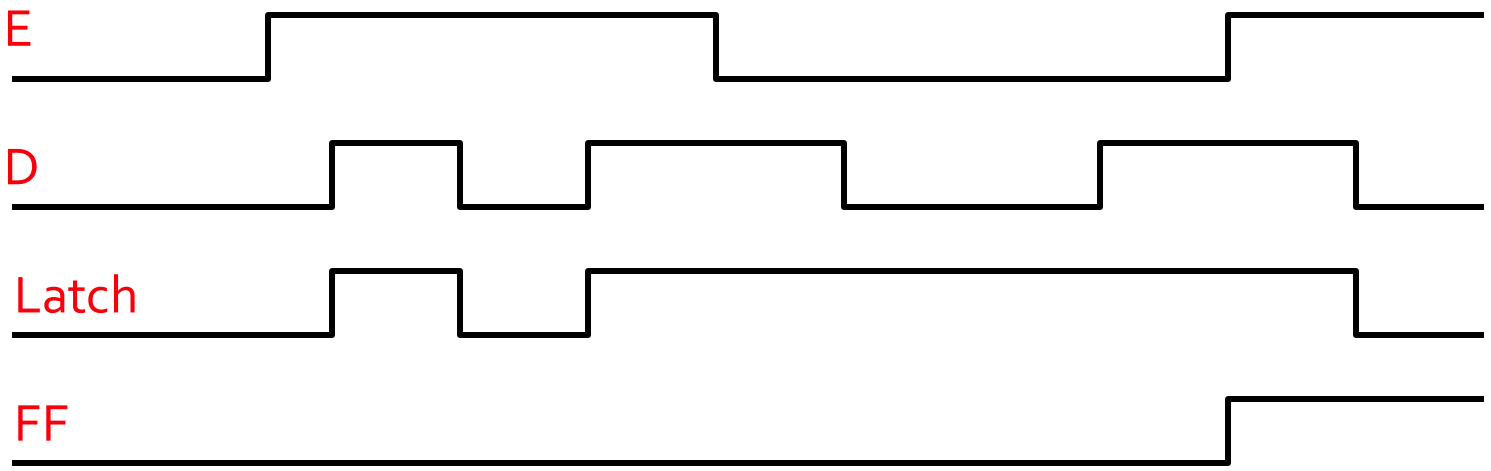
- Combinational Circuits
  - Assign statement
  - Always blocks
    - Case statement
  - Generator
- Sequential Circuits
  - Latches and FlipFlops
  - Always block
    - Sensitivity list
  - Blocking vs Nonblocking

Sequential Logic

- Output is a function of both the current inputs and the state.
- State represents the memory.
- State is a function of previous inputs.
- In synchronous digital systems, state is updated on each clock tick.

Timing: Level vs. Edge

- Latch timing
- Flip-flop timing

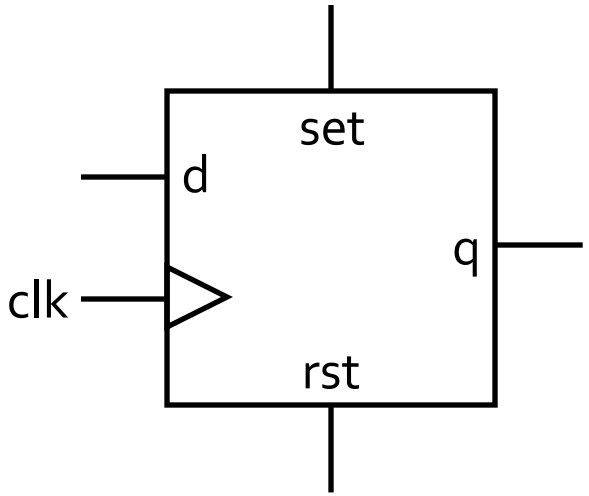


State Elements in Verilog

- Always blocks are the only way to specify the "behavior" of state elements.
- Synthesis tools will turn state element behaviors into state element instances.

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

    always @(posedge clk) begin
        if (rst)
            q <= 1'b0;
        else if (set)
            q <= 1'b1;
        else
            q <= d;
    end
endmodule
```



The Sequential `always` Block

```
module comb(input a, b, sel,
            output reg out);

    always @(*) begin
        if (sel) out = b;
        else out = a;
    end

endmodule
```

```
module seq(input a, b, sel, clk,
           output reg out);

    always @(posedge clk) begin
        if (sel) out <= b;
        else out <= a;
    end

endmodule
```

Blocking vs. Nonblocking Assignments

- Verilog supports two types of assignments within always blocks, with subtly different behaviors.
- Blocking assignment (=): evaluation and assignment are immediate

```
always @(*) begin
    x = a | b;          // 1. evaluate a|b, assign result to x
    y = a ^ b ^ c;      // 2. evaluate a^b^c, assign result to y
    z = b & ~c;         // 3. evaluate b&(~c), assign result to z
end
```

- Nonblocking assignment (<=): all assignments deferred to end of simulation time step after all right-hand sides have been evaluated (even those in other active always blocks)

```
always @(*) begin
    x <= a | b;         // 1. evaluate a|b, but defer assignment to x
    y <= a ^ b ^ c;     // 2. evaluate a^b^c, but defer assignment to y
    z <= b & ~c;        // 3. evaluate b&(~c), but defer assignment to z
                        // 4. end of time step: assign new values to x, y, and z
end
```

Combinational & Sequential

Use Nonblocking for Sequential Logic!

- **Blocking** VS **non-blocking** assignments
  - `=` - blocking
  - `<=` - non blocking, used only in always@(posedge) blocks

Blocking

Statements inside always block are executed **serially (like in software)**
Order Matters!

```
always @(*) begin
    b = a;
    c = b;
    d = b;
end
```

Is d == a ?

Non Blocking

Statements inside always block are executed in **parallel**. Value assigned to LHS are taken from "previous cycle" values of RHS

```
always @(posedge clk) begin
    b <= a;
    c <= b;
    d <= b;
end
```

Is d == a ?

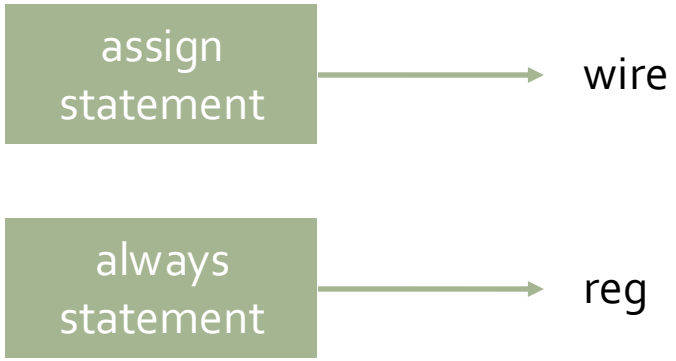
Simplified Verilog Guidelines

- Combinational logic:
  - Continuous Assignment:

    ```
    assign a = b & c;
    ```
  - Always block with @(*)

    ```
    always @(*) begin
            a = b & c; // blocking statement
    end
    ```
- Sequential logic:
  - Always block with @(posedge clk)

    ```
    always @(posedge clk) begin
            a <= b & c; // nonblocking statement
    end
    ```



Verilog in This Course

- We use behavioral modeling
- Favor continuous assign and avoid always blocks unless:
  - No other alternative: ex: state elements, case
  - Helps readability and clarity of code: ex: large nested if else
- Use named ports.
- Verilog is a big language. This is only an introduction.
  - Be careful about what you read on the web or hear from a chatbot. There are many bad examples out there.

Verilog Tips

- Do not write C-code
  - Think hardware, not algorithms
    - Verilog is inherently parallel
    - Compilers don't map algorithms to circuits well
- Do describe hardware circuits
  - First draw a dataflow diagram
  - Then start coding

Final Thoughts

- A large part of digital design is knowing how to write Verilog that gets you the desired circuit.
- First understand the circuit you want then figure out how to code it in Verilog.
- If you try to write Verilog without a clear idea of the desired circuit, you will struggle.

Common Errors

- **No clock** signal or wrong **reset level**
- Combinational-logic If-else, case statements **without default case** (this will generate **latches**).
- **No initial values** for registers (which do require it)
  - [need to set the initial values in reset condition] for control registers
- Blocking assignment (=) in `@(posedge clk)` blocks, instead of (<=), when Flip-flops flops are intended

Common Errors

- **Width mismatch on assignments between LHS and RHS**
- Counter **overflow/underflow**

Common Errors

- **Multi-drive**: Try to assign different values to a same reg/wire

Wrong

```
always @(posedge clk)
begin
    if (rst)
        q <= 1'b0;
    if (en)
        q <= d;
end
```

Correct

```
always @(posedge clk)
begin
    if (rst)
        q <= 1'b0;
    else if (en)
        q <= d;
end
```