RTL Coding Style

Summarized by Mu Wen

Adapted from Digital Design (2012) by Dally & Harting

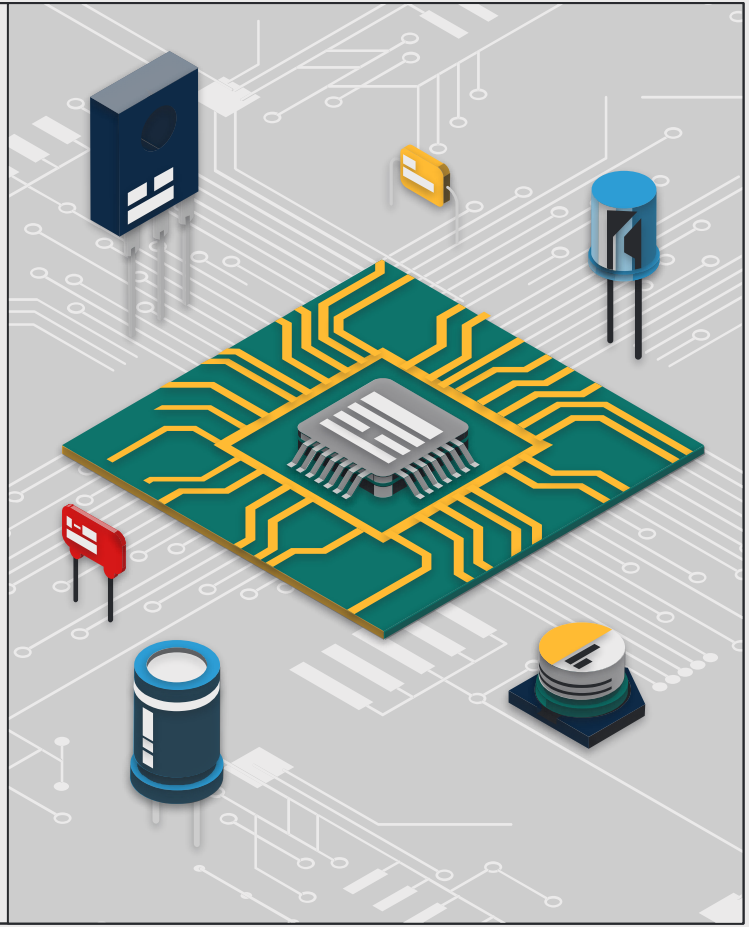
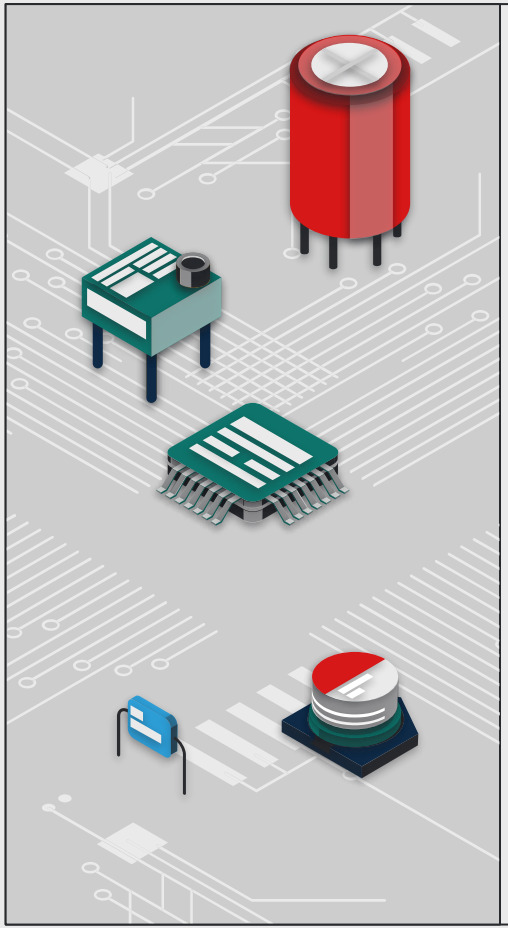


Table of contents



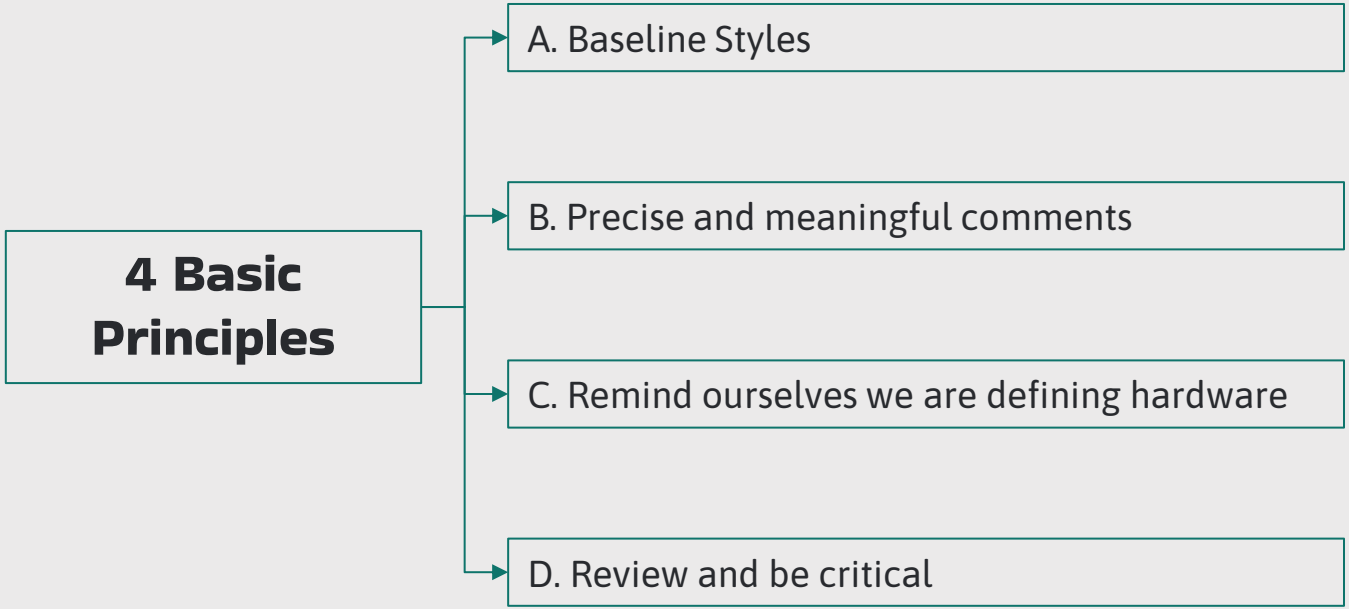
Introduction

This is **not a Verilog/System Verilog reference manual**, but aims serve as an optional guideline to help RTL designers write **maintainable** design codes.

- Why do we need this?
  - Most of the time, we work on legacy codes and we don't start things from scratch!
  - Imagine that it is your design and you have to revisit the code after a few months!

And the most important thing: we want to be **Great** RTL designers!

Recommended Principles



A. Baseline Styles

1. **Know where you state is**
   - Understand differences between combinational and sequential logics
   - This makes it easier to detect unintentional inferred latches
2. **Make your code readable**
   - Keep in mind that a design will probably be revised in future
   - Avoid frustration for future revisit
3. **Be defensive**
   - Keep thinking about what could break or go wrong
   - Include assertions for corner/edge cases
4. **Understand what your module will synthesize to**
   - Try to predict our code will be described structurally
   - This could help manual ECO and timing path fixes
5. **Understand what the synthesizer can't do**
   - They are not good at high-level optimizations and pipelining long paths
   - Try to make the designs modular to guide the tool

B. Precise and meaningful comments

- Well written codes should have **high-quality** comments.
- Gives **big picture** of what the code is doing
- Captures designer's **intent**
- Gives a **rationale** for design choices

Don'ts	Dos
Do not write comments that don't add any information ```verilog assign c = a + b; //add a and b ```	Write comments that add values ```verilog //factor the adder/subtractor out of the case statement //because the synthesizer won't combine two adders //Increments when down=0, decrements when down=1 assign outpm1 = out + {{(n-1){down}}, 1'b1}; ```

C. Remind ourselves we are defining hardware

- Although Verilog/System Verilog **code constructs** look very much like C/C++ codes, **do not fall in to a dangerous trap** of thinking we are writing a software program

Software Language	Hardware Language
One statement happens at a time (in sequence)	All statements happens at once (in parallel)

"Never forget that it is hardware – and everything happens at the same time"

— Dally & Harting

D. Review and be critical

- Similar to writing in English, one get better by reading/reviewing the works of others, being critical of their style, and **emulating styles that work**
- We can become a better RTL designer in the same way, **read and review RTL codes** of others whenever we are capable of
- **Be critical** of the code, point out what is both good and bad in our own code and in the code of others, **learn from good codes** that are reader friendly and easy to maintain
- Don't be defensive to improvement, **be open** to criticism, listen, and learn
- Good designer community **foster an engineering culture** that encourages designers to read and critique each other's codes through code review

**"Constructively judge the code,
not the person who produced it"**

— Dally & Harting

Recommended Principles

8 Rules for maintainability

1. All states should be in explicitly declared registers
2. Define reader-friendly combinational modules
3. Assign all variables under all conditions
4. Keep the designs modular (manageable size)
5. Keep the large modules structured
6. Use descriptive signals names
7. Use symbolic names for subfields of signals
8. Define constants

1. All states should be in explicitly declared registers

- All states in our design should be in **explicitly instantiated** registers or flops modules
- This helps us to understand what our RTL will be synthesized to

Don'ts	Dos
This bad style **causes confusion** on what is the current state and next state ```sv module BadCounter (clk, rst, out); parameter COUNTER_WIDTH = 4; input logic rst; input logic clk; output logic [COUNTER_WIDTH-1:0] out; always_ff @(posedge clk) begin out <= rst? '0: out+1; end endmodule //BadCounter ```	```sv module GoodCounter (clk, rst, out); parameter COUNTER_WIDTH = 4; input logic rst; input logic clk; output logic [COUNTER_WIDTH-1:0] out; always_ff @(posedge clk) begin out <= rst? '0: out+1; end endmodule //GoodCounter ```

2. Define reader-friendly combinational modules

- Modern synthesis tools are very good at optimizing small combi modules
- Utilize *always_comb* and *unique case* actively for readability
- Some sample codes are shown below

```sv
logic [NUM_DAYS_WIDTH-1:0] days;
logic [MONTH_WIDTH-1:0]      month;

always_comb begin: month_to_days
    //Sept, April, June, and Nov – 30 days
    //all the rest – 31 days
    //except for Feb – 28 days (assume same across all years for simplicity in this example)
    case (month):
        4'h4,
        4'h6,
        4'h9,
        4'hB: days      = 5'd30;
        4'h2: days      = 5'd28;
        default: days   = 5'd31;
    endcase
end //month_to_days
```

Example 1: Implement combinational functions that are best described with a table using a case statement.

```sv
logic [INPUT_WIDTH-1:0]        in;
logic [NUM_ONES_WIDTH-1:0] number_of_ones;


assign number_of_ones = in[0] + in[1] + in[3] + in[4];

```

Example 2: Combinational modules whose function is best described by an equation should just use an assign statement.

3. Assign all variables under all conditions

- This is intended to prevent undefined states or unintentional inferred latches for any conditional branches

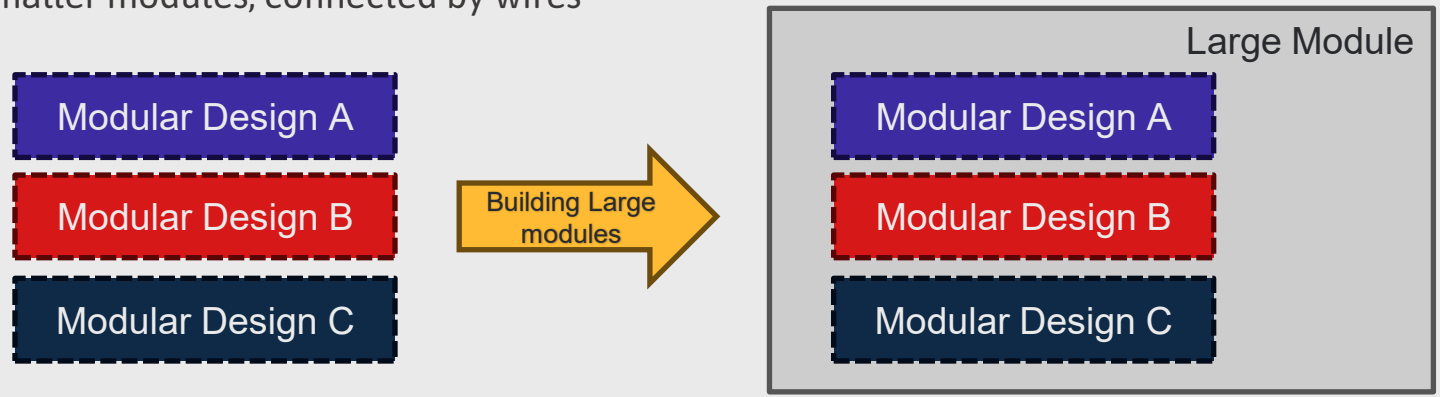
Don’ts	Dos
Bad code: a latch in unintentionally inferred because next is not assigned when rst is 0. ```sv logic [7:0] next; always_comb begin: sample_bad_cases if(rst == 1) next = ‘0; end //sample_bad_cases ```	Using a casex statement to assign a value to the next write based on four prioritized input signals. ```sv parameter DATA_WIDTH = 8; logic [DATA_WIDTH-1:0] next, indata; logic rst, load, shl, inc; always_comb begin: sample_good_cases casex({rst, load, shl, inc}) 4’b1xxx: next = ‘0; 4’b01xx: next = indata; 4’b001x: next = state<<1; 4’b0001: next = state+1; default: next = state; endcase end //sample_good_cases ```

4. Keep the designs modular (manageable size)

- To ensure readability and easier to predict what the module will be synthesized to
- This may also help if we were to implement unit testing

5. Keep the large modules structured

- By making use of rule #4, we can keep a large module by instantiations of many other smaller modules, connected by wires



6. Use descriptive signals names

- Our code will be much more readable if the signal names describe their functionality
- However, try to keep the names meaningful but not excessively long to enhance readability

Don’ts	Dos
Bad code: non-descriptive signal names `assign i = j << k;`	Using meaningful names to describe the functionality of signals. Difference design houses may use different conventions `assign aligned_mantissa = mantissa << exponent_difference;` If abbreviation is common and consistent through the whole design, we may also use the for readability: `assign algn_mts= mts << exp_diff;`

7. Use symbolic names for subfields of signals

- In most cases, signals with many bits can be broken into a number of subfields
- Our code becomes much more readable when we use symbolic names for these subfields
- We may also use the typedef in system Verilog if supported, consider the 32-bits example:

Don’ts

Bad code: splitting a packet with separate assignment statements per subfield is less readable and prone to errors. This is also not friendly for maintainability and scalability.

```sv
logic [31:0] in_req_packet;

logic [5:0] opcode          = req_packet[31:26];
logic [4:0] reg_a           = req_packet[25:21];
logic [4:0] reg_b           = req_packet[20:16];
logic [4:0] reg_c           = req_packet[15:11];
logic [4:0] sa              = req_packet[10:6];
logic [5:0] func            = req_packet[5:0];
```

Dos

Better for readability and maintainability, examples of using typedef struct packed

```sv
typedef struct packed {
    logic [5:0] opcode          = req_packet[31:26];
    logic [4:0] reg_a;          = req_packet[25:21];
    logic [4:0] reg_a;          = req_packet[20:16];
    logic [4:0] reg_a;          = req_packet[15:11];
    logic [4:0] sa;             = req_packet[10:6];
    logic [5:0] func;
} req_packet_struct

logic [FULL_PACKET_WIDTH-1:0] in_req_packet;
req_packet_struct                 req_packet;

assign req_packet.opcode        = in_req_packet[31:26];
assign req_packet.reg_a         = in_req_packet[25:21];
assign req_packet.reg_b         = in_req_packet[20:16];
assign req_packet.reg_c         = in_req_packet[15:11];
assign req_packet.sa            = in_req_packet[10:6];
assign req_packet.func          = in_req_packet[5:0];
```

8. Define constants

- Hardcoded numbers should rarely appear in our Verilog code
- By using `defined *constant* or typedef, our codes become much more readable and scalable

Don’ts

Bad code: constants are hardcoded and not meaningful

```
logic [8:0] opcode;
logic [8:0] func;
logic [2:0] alu_op;

always_comb begin: decode_cases
    unique casez({opcode, func})

        {8'h00, 8'h20}: alu_op = 3'h5;
        {8'h00, 8'h21}: alu_op = 3'h6;
        ...
        {8'h23, 8'h??}: alu_op = 3'h5;
        ...
    endcase
end //decode_cases
```

Dos

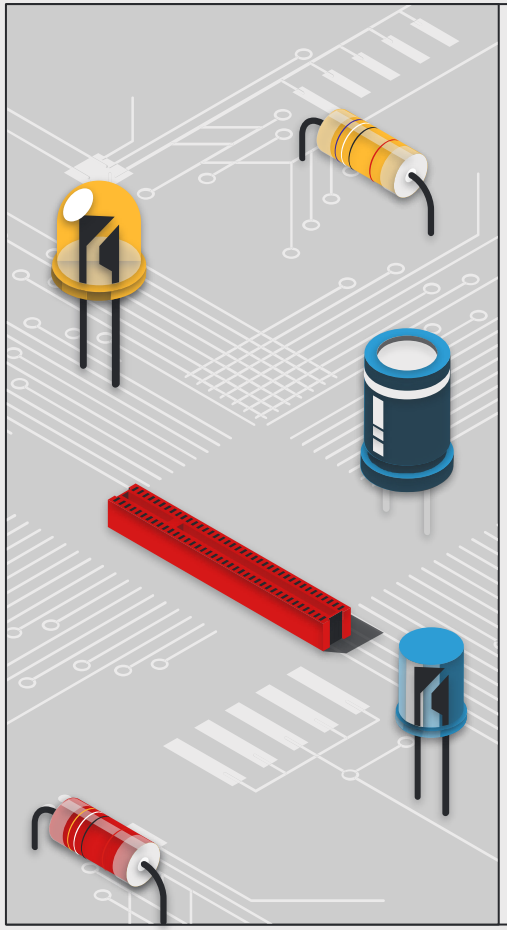
Better for readability, examples of using defines of typedef enum

```
`define ADD_OP 3'h5
`define SUB_OP 3'h6

logic [ALU_WIDTH-1:0] alu_op;
typedef enum logic [OPCODE_WIDTH-1:0] {
    RTYPE_OPC = 8'h00,
    LW_OPC    = 8'h23,
    ...
} opcode_type;
typedef enum logic [FUNC_WIDTH-1:0] {
    ADD_FUNC = 8'h20,
    SUB_FUNC = 8'h21,
    ...
} func_type;

opcode_type opcode;
func_type   func;

always_comb begin: decode_cases
    unique casez({opcode, func})
        {RTYPE_OPC, ADD_FUNC}: alu_op = 3'h5;
        {RTYPE_OPC, SUB_FUNC}: alu_op = 3'h6;
        ...
        {LW_OPC,    8'h??}: alu_op = 3'h5;
        ...
    endcase
end //decode_cases
```



Thanks!

mwchuan@outlook.com

Mu Wen Chuan

Created using template from