Synthesizable Verilog Constructs — Syntax Reference

Below is a compact **synthesizable RTL coding reference** showing the correct syntax for each major construct. These are the forms you must use in real RTL for ASIC/FPGA design.

---

1. Module and Ports

```
module my_module (
   input  wire       clk,
   input  wire       reset,
   input  wire [7:0]  a,
   input  wire [7:0]  b,
   output reg  [7:0]  y
);
// body here
endmodule
```

---

2. Parameters

```
module adder #(parameter WIDTH = 8) (
   input  wire [WIDTH-1:0] a, b,
   output wire [WIDTH-1:0] sum
);
   assign sum = a + b;
endmodule
```

---

3. typedef enum (SystemVerilog) for FSM States

```
typedef enum logic [1:0] {
   IDLE  = 2'b00,
   LOAD  = 2'b01,
   PROC  = 2'b10,
   DONE  = 2'b11
} state_t;

state_t state, next_state;
```

(For plain Verilog use `parameter` for state encoding.)

---

4. Always Block (Combinational)

```
always @(*) begin
```

```
    out = 8'h00;            // default assignment
    if (enable)
        out = a & b;        // blocking (=) for comb
    else
        out = 8'hFF;
end
```

---

5. Always Block (Sequential / Flip-Flop)

- **Synchronous Reset**

```
always @(posedge clk) begin
    if (reset)
        q <= 0;
    else
        q <= d;             // non-blocking (<=) for seq
end
```

- **Asynchronous Reset**

```
always @(posedge clk or posedge reset) begin
    if (reset)
        q <= 0;
    else
        q <= d;
end
```

---

6. If / Else

```
always @(*) begin
    if (sel == 2'b00)
        y = a;
    else if (sel == 2'b01)
        y = b;
    else
        y = 8'h00; // default
end
```

---

7. Case Statement

```
always @(*) begin
    y = 8'h00; // default
    case (state)
        IDLE : y = 8'h00;
        LOAD : y = a;
        PROC : y = a + b;
        DONE : y = 8'hFF;
        default: y = 8'h00; // handles illegal states
    endcase
end
```

---

8. Assign (Continuous Assignment)

```
assign sum = a + b;
assign flag = (a == b);
```

---

9. Generate Block

```
genvar i;
generate
    for (i = 0; i < WIDTH; i = i + 1) begin : gen_loop
        assign y[i] = a[i] ^ b[i];
    end
endgenerate
```

---

10. Memory Declaration

```
reg [7:0] mem [0:255];  // 256x8 RAM

always @(posedge clk) begin
    if (we)
        mem[addr] <= data_in;   // write
    data_out <= mem[addr];      // sync read
end
```

---

11. For Loop (inside always/generate)

```
integer i;
always @(*) begin
    sum = 0;
    for (i=0; i<8; i=i+1)
        sum = sum + data[i];
end
```

---

12. Finite State Machine (Full Example)

```
// State Register
always @(posedge clk or posedge reset) begin
   if (reset)
      state <= IDLE;
   else
      state <= next_state;
end

// Next-State Logic
always @(*) begin
   next_state = state;
   case (state)
      IDLE: if (start) next_state = LOAD;
      LOAD:          next_state = PROC;
      PROC:          next_state = DONE;
      DONE:          next_state = IDLE;
      default:       next_state = IDLE;
   endcase
end
```

```
//==============================================================
// RTL Skeleton Example
//==============================================================

module rtl_skeleton #(
   parameter WIDTH = 8,              // parameter example
   parameter DEPTH = 16
)(
   input  wire              clk,     // clock
   input  wire              reset,   // reset
   input  wire              start,   // control
   input  wire [WIDTH-1:0]  a, b,    // datapath inputs
   output reg  [WIDTH-1:0]  y,       // datapath output
   output reg               done
);

   //==========================================================
   // 1. Typedef Enum for FSM States
   //==========================================================
   typedef enum logic [1:0] {
      IDLE  = 2'b00,
      LOAD  = 2'b01,
      PROC  = 2'b10,
      DONE  = 2'b11
   } state_t;

   state_t state, next_state;

   //==========================================================
   // 2. Internal Registers, Memory, and Wires
   //==========================================================
   reg [WIDTH-1:0] reg_a, reg_b;
   reg [WIDTH-1:0] mem [0:DEPTH-1];   // simple RAM
   wire [WIDTH-1:0] sum;              // combinational
   integer i;

   // Continuous assignment
   assign sum = reg_a + reg_b;

   //==========================================================
   // 3. State Register (Sequential Always)
   //==========================================================
   always @(posedge clk or posedge reset) begin
      if (reset)
         state <= IDLE;
      else
         state <= next_state;
   end
```

```
//==========================================================
// 4. Next-State Logic (Combinational Always + Case)
//==========================================================
always @(*) begin
   next_state = state;      // default
   case (state)
      IDLE : if (start) next_state = LOAD;
      LOAD :            next_state = PROC;
      PROC :            next_state = DONE;
      DONE :            next_state = IDLE;
      default:          next_state = IDLE;
   endcase
end

//==========================================================
// 5. Output and Control Logic (Combinational Always)
//==========================================================
always @(*) begin
   // defaults to avoid latches
   y    = 0;
   done = 0;

   case (state)
      IDLE : begin
         y    = 0;
         done = 0;
      end
      LOAD : begin
         y    = a;     // capture input
      end
      PROC : begin
         y    = sum;   // combinational result
      end
      DONE : begin
         done = 1;
      end
      default : begin
         y    = 0;
         done = 0;
      end
   endcase
end
```

```
    //==========================================================
    // 6. Datapath Registers (Sequential Always + If/Else)
    //==========================================================
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            reg_a <= 0;
            reg_b <= 0;
        end else begin
            if (state == LOAD) begin
                reg_a <= a;
                reg_b <= b;
            end
        end
    end

    //==========================================================
    // 7. Simple RAM (Synchronous Read/Write)
    //==========================================================
    always @(posedge clk) begin
        if (state == LOAD)
            mem[0] <= a;          // write
        if (state == PROC)
            y <= mem[0] + b;      // read and compute
    end

    //==========================================================
    // 8. Generate Block (Optional Replication)
    //==========================================================
    genvar gi;
    generate
        for (gi = 0; gi < WIDTH; gi = gi + 1) begin : gen_xor
            wire temp;
            assign temp = a[gi] ^ b[gi];
            // Can be used in datapath (example only)
        end
    endgenerate

endmodule
```