Format of module:
module *module-name*(input *input1-name*, input *input2-name*, […],
output *output1-name*, […]);
declarations
statements
endmodule

Declarations:
wire *wire-name*; - declares internal wire
reg *reg-name*; - Verilog register; will hold value until next assignment
wire [*n*:0] *wire-name*; - declares n-bit wire
reg [*n*:0] *reg-name*; - declares n-bit register

parameter *PARAM* = *value* - declares constant value

Constants examples:
3'b010 - 3-bit binary value "010"
8'hA9 - 8-bit hex value "A9"
6'd35 - 6-bit decimal value "35"

Statements:

assign *l-value* = *expression*;
- This implies combinational logic

always @(*sensitivity list*)
begin
non-blocking statements
end
- This implies combinational or sequential logic

Some basic operators:

~	NOT
&	AND
|	OR
^	XOR
==	equals
!=	not equals
~&, ~|, ~^	NAND, NOR, XNOR

Example: 4-bit inverter

```
module inverter(input [3:0] in, output [3:0] out);
   assign out = ~in;
endmodule
```

Logic operators can be used as unary operators.
Example: 4-input AND gate

```
module and4 (input [3:0] in, output out);
   assign out = &in;
endmodule
```

Always blocks

- evaluated whenever signal in sensitivity list changes
- should use non-blocking statements, not blocking:
    - q <= d; Good!
    - q = d; Bad!

Non-blocking statements:

Assignment: *l-value* <= *expression*;

Conditional:
```
if (expression)  statement;
else if(expression) statement;
else statement;
```

Case:
```
case (a)
   2'b00: statements;
   2'b01: statements;
   2'b10: statements;
   2'b11: statements;
endcase
```

Example: edge-triggered flip-flop

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

   always @(posedge clk)
      q <= d;
endmodule
```

Example: flip-flop with asynchronous reset

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

   always @(posedge clk, posedge reset)
      if (reset) q <= 1'b0;
      else       q <= d;
endmodule
```

Example: 2-to-4 decoder

```
module decoder(input [1:0] a, output reg [3:0] z);

  always @(*)
    case(a)
       2'b00: z <= 4'b0001;
       2'b01: z <= 4'b0010;
       2'b10: z <= 4'b0100;
       2'b11: z <= 4'b1000;
    endcase
endmodule
```

Sequence detector

```
module detect_010(input clk, input reset, input a,
                  output z);

   reg [1:0] state, nextstate;

   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;
   parameter S3 = 2'b11;

   // State register
   always @(posedge clk, posedge reset)
     if (reset)   state <= S0;
     else         state <= nextstate;

   // Next state logic
   always @(*)
        case (state)
              S0: if (a)  nextstate <= S0;
                  else   nextstate <= S1;
              S1: if (a)  nextstate <= S2;
                  else   nextstate <= S1;
              S2: if (a)  nextstate <= S0;
                  else   nextstate <= S3;
              S3: if (a)  nextstate <= S2;
                  else   nextstate <= S1;
        endcase

   // Output logic
   assign z = state[1] & state[0];
endmodule
```

Style guidelines:

- Use only nonblocking assignments inside always blocks.
- Define combinational logic using *assign* statements when practical. Only use always blocks for combinational logic if using *if* or *case* statements will make your logic clearer or more compact.
- When modeling combinational logic with an *always* block, if a signal is assigned in a branch of an *if* or *case* statement, it must be assigned in all branches.

  Proper use of *if*: (describes a mux)
  ```
  if (s)   z <= a;
  else     z <= b;
  ```

  Bad use of *if*: what value does z have if s = 0?
  ```
  if (s)  z <= a;
  ```

- Use parameters to define state names and constants.
- Indent, comment, use meaningful signal names.