Below is a list of coding styles that might cause synthesis problems and/or inefficient design implementation. This is not an exhaustive list, but it contains the most common problems people have had in the lab (warnings given are from FPGA Express – the synthesis tool we use).

1. **Incomplete Sensitivity List**
   When using an always statement to implement combinational logic, the sensitivity list has to include all signals that appear on the right hand side of the assignments inside the always block.
   
   **Incorrect:**
   ```
   always @(a or b or c)
   begin
     x <= a + b;
     y <= b + c + d; // might cause latch inferred warning in FPGA
                    // Express
   end
   ```
   
   **Solutions:**
   - Put signal d in the sensitivity list in the always statement
   - Use assign statement to implement combinational logic.

   **Correct:**
   ```
   always @(a or b or c or d)
   begin
     x <= a + b;
     y <= b + c + d;
   end
   ```

2. **The Use of ‘case’ vs. ‘if – else’ Statements**
   In general, case statements would translate to parallel muxes in hardware. Meanwhile, the use of if – else statements results in priority-based hardware, which can slow down the overall implementation of the design. Thus, only use if – else statement when priority-based hardware is required. Otherwise, use case statements to avoid possible inefficient implementation of the design.

   While unlikely to produce incorrect outputs, the slowdown caused by large priority-based implementations can cause problems for projects with strict clock frequency targets.
3. **Incomplete ‘case’ Statement**

To avoid problems with inferred latches, always specify all the possible cases in a case statement.

Incorrect:

```vhdl
input [1:0] select
always @ (a or b or c or d or select)
begin
  case (select)
    0: x <= a;
    1: x <= b;
    2: x <= c;
  end
end
```

Solutions:
- Completely specify the cases in the case statement
- Always use default statement.

Correct:

```vhdl
input [1:0] select
always @ (a or b or c or d or select)
begin
  case (select)
    0: x <= a;
    1: x <= b;
    2: x <= c;
    3: x <= d; // complete cases based on the select signal
    default: x <= 1'bx; // add default statement to prevent unwanted latches and/or aid debugging
  end
end
```

4. **Unspecified Assignments in ‘case’ or ‘if – else’ Statements**

Incorrect:

```vhdl
if (....)
begin
  a <= 1;
  b <= 0;
end
else if (....)
begin
  a <= 0;
end
else
begin
  c <= 1;
end
```

```vhdl
// latch inferred for the signal c
// latch inferred for the signal b
// latch inferred for the signals a and b
```
Solutions:

- Always assign known values to all the signals in a case or if-else statements.

Correct:

```plaintext
if (....)
begin
    a <= 1;
    b <= 0;
    c <= 0;
end
else if (....)
begin
    a <= 0;
    b <= 0;
    c <= 0;
end
else
    begin
        a <= 0;
        b <= 0;
        c <= 1;
    end
```

Note: the piece of code above is typically found in control modules, thus the designer knows what values each signal has to be in each case/if/else blocks. This problem will not usually occur if the only item in the sensitivity list for the always block is a clock signal.

5. **Signal Assignment in Multiple ‘always’ Blocks**

Multiple always blocks may not assign values to the same signal in the same time step. This would result in the signal having an indeterminate value. FPGA express would produce an “assignment in multiple always blocks” warning.

Incorrect:

```plaintext
always @ (posedge clk)
begin
    a <= 1;
    b <= 0;
end

always @ (posedge clk or reset)
begin
    a <= 0;  // multiple assignment to signal a
    c <= 1;
end
```

Solutions:

- Only one behavioral block can have an assignment to a signal.
- The use of assign statements for combinational logic.
Correct:
always @ (posedge clk)
begin
    a <= 1;
    b <= 0;
end

always @ (posedge clk or reset)
begin
    c <= 1;
end

6. **Dangling Wires**

Due to misspelling (very common!) or unused signals, there might be dangling wires in the design. The synthesis tool might optimize these wires away. Further, due to the recursive algorithm used to optimize the design synthesis, the logic inside the design itself might get optimized away. When trying to do post-synthesis simulation, your design will not generate correct results since part of the logic or even all the logic in the design gets optimized away, in which case you are left with just input and output ports with no logic in between.

Incorrect:
Module Incomplete (a, b, out);
input    a;
input    b;
output   out;
assign Out = a ? b : 1'b0;                // output port out is not being driven here
endmodule

Solutions:
- Check for misspelling
- Check for dangling or unconnected wires
- Check for incorrect assignments (i.e. assigning to incorrect output signal).

Correct:
Module Incomplete (a, b, out);
input    a;
input    b;
output   out;
assign out = a ? b : 1'b0;
endmodule
References: (all reference material is linked from the class website at: http://www.caewisc.edu/~ece554/)

HDL Synthesis References (hardcopies available in the lab)
Verilog – FPGA Express Synthesis Reference:
http://www.caewisc.edu/~ece554/website/ToolDoc/FPGA_Express_3.6.1_docs/hdlref.pdf

VHDL – FPGA Express Synthesis Reference:
http://www.caewisc.edu/~ece554/website/ToolDoc/FPGA_Express_3.6.1_docs/vhdlref.pdf

Hints and Tips
http://www.caewisc.edu/~ece554/website/ToolDoc/FPGA_Express_3.6.1_docs/vendordocs/Xilinx_hdl-coding.pdf

Papers on Design Synthesis
SNUG 2000 (San Jose): Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!

SNUG 1999 (San Jose): RTL Coding Styles That Yield Simulation and Synthesis Mismatches

SNUG 1998 (San Jose): State Machine Coding Styles for Synthesis

The papers listed above can be found at: http://www.sunburst-design.com/papers/