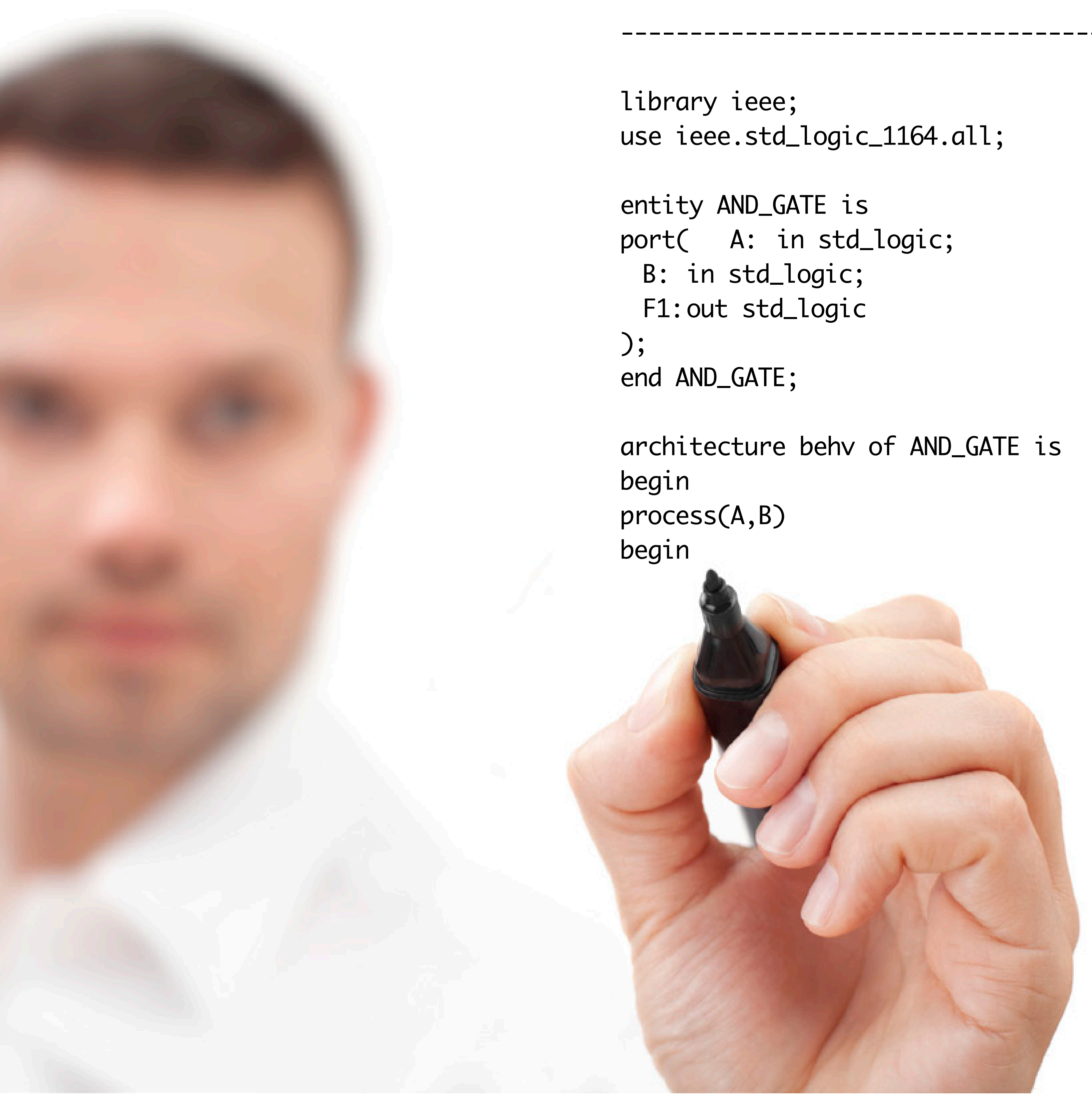
What's in a Name?



```
------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity AND_GATE is
port(   A: in std_logic;
  B: in std_logic;
  F1:out std_logic
);
end AND_GATE;

architecture behv of AND_GATE is
begin
process(A,B)
begin
```

Dave Vandenbout
XESS Corp. - Founder

When you're writing your HDL code, what consideration should be top-of-mind?

Creating a good hierarchy?

Maintaining a synchronous design?

Registering inputs and outputs?

I'll suggest a more basic concern:

finding good names for stuff.

Think for a moment what your HDL coding life would look like if you took the time to clearly name the various I/Os, signals, registers, modules, etc.

• You would write better documentation.

The HDL code itself would help to tell the story about how the design does what it is meant to do. This means the surrounding comments can concentrate on telling why the design is built this way.

• You would write less explicit documentation.

Because the HDL names assist with the documentation, fewer actual comments are needed. And these comments have to be updated less frequently as the design changes because the "why" of a design changes less frequently than "how" it actually operates.

• You would introduce fewer bugs.

Bugs get into the code because we place them there, usually because we're confused about what the design is doing. Good names carry along the meaning of the problem the design is meant to solve, so it's easier to load the problem into your head. Then you'll spend less mental energy translating the variables back into the problem domain and more on producing correct code.

• Your debugging sessions would be easier.

For the same reason, it's easier to trace and find errors when the debugger shows variables whose names refer directly to items in the problem domain.

• Your designs would be re-used more often.

If a design is easier for you to understand and modify, the same will apply to others and they'll be more likely to use it as well.

Here's another indication of the importance of naming: "The Power of Variable Names" in Code Complete, is 25% longer than any other chapter. You could stop reading this right now and go read that chapter, but I'll synopsize the germane points for you:

• A variable name should describe what it represents.

For example, heightOfAscent would be a good name for a variable in a telemetry module that records the current altitude of a rocket. Not so for a variable named h or (even worse) x.

• A variable should refer to the problem domain, not the implementation.

For example, naming a variable heightCounter implies that the rocket's altitude is maintained within a counter. This speaks to how the altitude is computed within the circuit, but that may change as the design's implementation changes. You don't want to have to change your variable names if your logic changes or – worse yet – have your names give misleading information about how the design works.

• Variable names should be between 10 and 16 characters.

This makes the variables easiest to comprehend while still conveying meaning (although you can stretch this to 8-20 chars with only slightly worse results). Of course, variable names that describe the problem domain can get rather long (heightOfAscent is already at 14 characters), so you'll have to employ some techniques to shorten them like removing nonleading vowels (hghtOfAscnt) and removing articles (hghtAscnt).

• The greater the scope of the variable, the more descriptive the name should be.

For example, you can use i as the index in a short generate loop but not for a 1000-line block of code (well, nothing would be appropriate for that).

In addition to the general principles shown above, I also have conventions for how I adorn names in my VHDL code. I use capitalization and append suffixes to make it easier and faster for me to generate meaningful, consistent names. It also indicates where the signals come from and where they can be used.

Here are the rules I use:

• Entities, architectures, procedures, functions, typenames: CamelCase with an initial uppercase letter.

• Packages: CamelCase with an initial uppercase letter and ending with Pckg.

• Component instantiations: CamelCase with an initial U.

• Constants & generics: all caps with underscores and either a _C or _G as a suffix.

• Signals & variables: CamelCase with an initial lowercase letter and one or more of the following suffixes:

- _i: Input port.

- _o: Output port.

- _s: Signal local to architecture.

- _v: Variable local to process.

- _b: Active-low (complementatry) signal.

- _r: Current register value.

- _x: Next register value after clock edge.

- _a: Asynchronous signal.

- _d: Delayed version of signal.

- _e: Enabled version of signal.

To show how I use my conventions, here's an artificial example of a module that integrates the difference of two signals:

The comments in the code show some of the places where my naming conventions help out. But there are also a couple of places where I violate my conventions:

• I use short, nondescriptive names for the a_i and b_i inputs. In my defense, there aren't any really good names for these since this is just a module for performing a general-purpose calculation that would be used in some larger application. I also tried to mitigate this by placing AminusB in the output names to show that the difference of these two inputs is what's being worked with.

• I violated the CamelCase naming format for some of the signals such as intgrlAminusB_r because the correct version, intgrlAMinusB_r, looked rather odd and was hard to read.

These violations demonstrate the last and most important naming convention: don't be a prig! These rules exist to serve you and not the other way around. If you find places where they make the code less clear, then either violate them or change the conventions to account for these new circumstances. There's no reason for slavish adherence to some standard if it generates poor code.

```
entity DiffIntgrtr is  -- shortened name for DifferenceIntegrator
    generic (
        GAIN_G : integer
        );
    port (
        clk_i           : in  std_logic;
        a_i             : in  integer := 0;
        b_i             : in  integer := 0;
        aMinusB_ao      : out integer;  -- asynchronous output of A - B
        intgrlAminusB_o : out integer   -- synchronous output of Integral(A-B)
        );
end entity;


architecture arch of DiffIntgrtr is
    signal aMinusB_s       : integer;  -- internal async. version of A - B
    signal intgrlAminusB_r : integer;  -- current value of Integral(A-B)
    signal intgrlAminusB_x : integer;  -- next value of Integral(A-B)
begin
    -- async subtractor module (in reality, would use the subtraction operator)
    USubtractor : Subtractor port map(
        minuend_ai    => a_i,        -- see how it's easy to tell
        subtrahend_ai => b_i,        -- the module's inputs from its
        diff_ao       => aMinusB_s   -- async output.
        );

    aMinusB_ao <= aMinusB_s;  -- output the internal version of the difference

    -- compute the next value of the diff integral
    process (aMinusB_s, intgrlAminusB_r)  -- combinatorial process
    begin
        intgrlAminusB_x <=                            -- notice that the _x/_r signal must
            intgrlAminusB_r + aMinusB_s * GAIN_G; -- appear on the LHS/RHS of the
    end process;                                      -- '<=' in a combinatorial process

    -- xfer the next value of the diff integral into the current value
    process (clk_i)  -- sequential process
    begin
        if rising_edge(clk_i) then                  -- notice that the _r/_x signal must
            intgrlAminusB_r <= intgrlAminusB_x; -- appear on the LHS/RHS of the
        end if;                                     -- '<=' in a sequential process
    end process;

    intgrlAminusB_o <= intgrlAminusB_r;  -- output the integrated difference

end architecture;
```

It can be hard to remember a new set of naming rules. To help myself, I created a bunch of macros for the Notepad++ editor which automatically generate VHDL that follows my naming conventions. While I don't recommend my rules for everyone, you should have some convention to guide you. Maybe you can modify my macros to fit your design environment. ■