# VHDL by Example, errata

```
-- ------------------------------------------------
-- Header information
-- ------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity bus_breakout is
  port (
         -- Inputs
         in_1   : in  std_logic_vector(3 downto 0);
         in_2   : in  std_logic_vector(3 downto 0);
         in_3   : in  std_logic;
         -- Outputs
         out_1  : out std_logic_vector(5 downto 0)
       );
end entity bus_breakout;

architecture bus_breakout_arch of bus_breakout is

begin
    --------------- Design implementation --------

    out_1 <= (   in_2(3 downto 2)
               & (in_1(3) AND in_2(1))
               & (in_1(2) AND in_2(0))
               & in_1(1 downto 0)
             );

end architecture bus_breakout_arch;
```

**Bus Breakout**

The “in_3” input port is not used here. This is allowed in VHDL, but can be confusing.

We now introduce a few common state-type operations to show how increasingly sophisticated register-based functions are implemented in process statements. A four-bit counter is enabled by a "start" event, and stopped by a "stop" event. The SR flop allows the start and stop events to be short, e.g. one-clock pulses, rather than a continuously enabling flag. Additionally, for further illustration, we delay the ~~start~~ stop signal two clocks and provide it as an output.

You'll notice that we have not shown the asynchronous reset in the diagram. This is done for clarity; from this point forward it is assumed. It is implemented in the code, and always will be (in this book).

P28

```
    end if;
  end process;

  count <= std_logic_vector(count_us);

end architecture clks_n_regs_5_arch;
```

## SR flop and counter, using variables

We are able to replace these signals with variables, since they are used exclusively within the process. Notice that the variable declarations are made within the process statement rather than the beginning of the architecture. You can see that within this process we use variables in the same way as their predecessor signals of clks_n_regs_4.vhd. There is one very key difference between signals and variables, however: a variable takes on its assignment immediately, while the signal assignment waits until the next rising clock edge. The difference is fundamental, and results in a different behavior in this example. When "count_us" is a signal, here's what the result is after "rst" goes low (assuming "start" is already set):

cnt_en

after 1st clock: count = "0000"

after 2nd clock: count = "0001"

This is because "cnt_en" must be set high with the first clock before the counter is enabled.

But using a variable for "count_us", here's what we get:

cnt_en



P33

previous

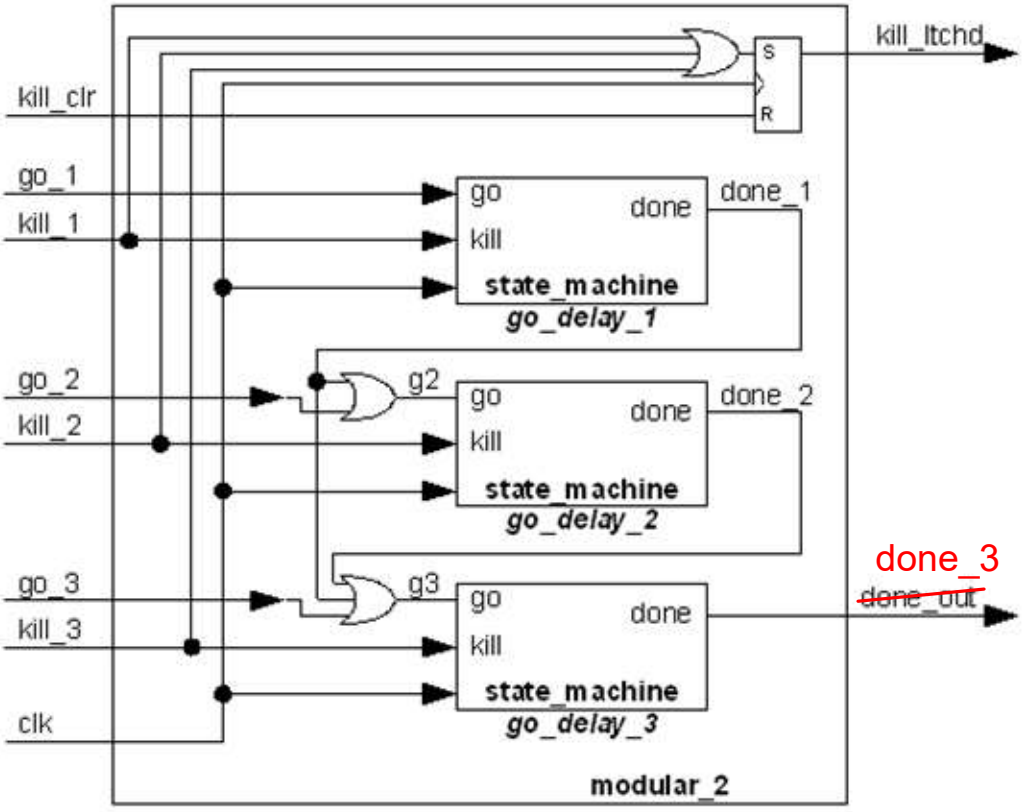
Finally, be aware that, as in the with/select ~~above~~, many synthesis programs require the "others" statement in the case list, even if the case statement already includes all possible selection branch combinations (considered "full").

The counter and output register of this module are similar to those we've already seen. Note that we decode state machine states directly in these blocks using the "state_reg" register signal and associated labels. Also note that I've set the clear conditions of the counter to match the diagram above (via the "Finish" and "Abort" states), but the following works equally well, and in fact, has the advantage of defining the state of the counter completely to include the "Idle" state (by default):

```
-- duration counter
if (state_reg = Active) then
   count <= count + 1;
else
   count <= X"00";
end if;
```

Finally, note that although the counter is seven bits, I've declared it as eight. This is a convenience so that I can use the shortened X"00" form of expression instead of "00000000". Since the eighth bit (the MSb) is not actually used, the synthesis tool will strip it away anyway.

As we've noted, VHDL code can be structured in a variety of ways. Some designers might prefer the auxiliary counter and output register to be collected into one process statement along with the state machine. This is how it might look:

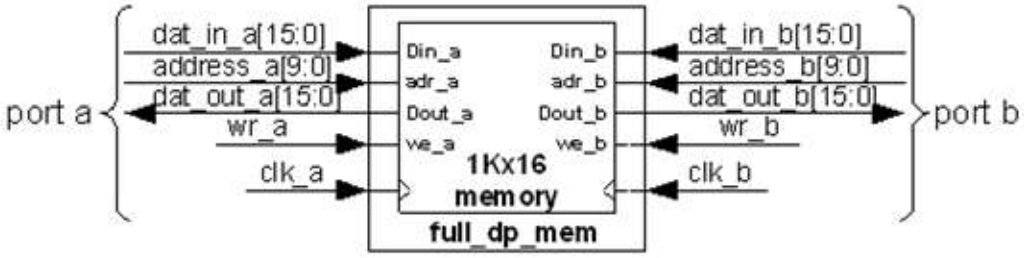



Second Modular Example

P50

address simultaneously. There are two possibilities: 1) the data that is read is the original value before the write replaced it (called "read-before-write"), or 2) the data is the new value that is being written (called "write-before-read"). Our code implies a write-before-read operation based on the simple fact that the write assignment comes before the read assignment in the process statement (reversing the order would imply a read-before-write). Newer RAM blocks typically accommodate either type of operation, but some older versions are fixed, and in that case, as the designer you would have to make sure your code matches (and that your design operates correctly).

Next, we look at a full dual-port memory, where both ports have both write and read capability.




**Full Dual-port Memory**

It should be noted that vendors have different memory capabilities, and generally specific methods of inferring RAM. This simple example of write-before-read will not work for many vendor synthesis tools, and the designer is advised to consult the vendor documentation for guidance.