5 A structured VHDL design method

5.1 Introduction

The VHDL language [22] was developed to allow modelling of digital hardware. It can be seen as a super-set of Ada, with a built-in message passing mechanism called *signals*. The language was defined in the mid-1980's as a respons to the difficulties of developing, validating and co-simulating increasingly complex digital devices developed within the VHSIC[23] program. The main focus was to be able to write *executable* specifications, and allow specifications (or models) from different providers (companies) to be simulated together.

When the language was first put to use, it was used for high-level behavioural simulation only. 'Synthesis' into VLSI devices was made by manually converting the models into schematics using gates and building blocks from a target library. However, manual conversion tended to be error-prone, and was likely to invalidate the effort of system simulation. To address this problem, VHDL synthesis tools that could convert VHDL code directly to a technology netlist started to emerge on the market in the begining of 1990's. Since the VHDL code could now be directly synthesised, the development of the models was primarily made by digital hardware designers rather than software engineers. The hardware engineers were used to schematic entry as design method, and their usage of VHDL resembled the dataflow design style of schematics. The functionality was coded using a mix of concurrent statments and short processes, each decribing a limited piece of functionality such as a register, multiplexer, adder or state machine. In the early 1990's, such a design style was acceptable since the complexity of the circuits was relatively low (< 50 Kgates) and the synthesis tools could not handle more complex VHDL structures. However, today the device complexity can reach several millions of gates, and the synthesis tools accept a much larger part of the VHDL standard. It should therefore be possible to use a more modern and efficient VHDL design method than the traditional 'dataflow' version. This chapter will describe such a method and compare it to the 'dataflow' version.

5.2 The problems with the 'dataflow' design method

The most commonly used design 'style' for synthesisable VHDL models is what can be called the 'dataflow' style. A larger number of concurrent VHDL statements and small processes connected through signals are used to implemenet the desired functionality. Reading and understanding dataflow VHDL code is difficult since the concurrent statements and processes do not execute in the order they are written, but when any of their input signals change value. It is not uncommon that to extract the functionality of dataflow code, a block diagram has to be drawn to indentify the dataflow and dependecies between the statements. The readability of dataflow VHDL code can compared to an ordinary schematic where the wires connecting the various blocks have been removed, and the block inputs and outputs are just labeled with signal names!

Below is a small (but real) example of dataflow code taken from the memory controller of a 32-bit processor. The memory controller contains approximately 2,000 of these small processes and concurrent statements, making it very difficult for an engineer that did not design to code to understand and maintain it:

```
CBandDatat_LatchPROC9F: process(MDLE, CB_In, Reset_Out_N)
 begin
  if Reset_Out_N = '0' then
     CBLatch_F_1       <= "0000";
  elsif MDLE = '1' then
     CBLatch_F_1       <= CB_In(3 downto 0);
  end if;
 end process;


 CBandDatat_LatchPROC10F: process(MDLE, CB_In, DParIO_In, Reset_Out_N)
 begin
  if Reset_Out_N = '0' then
     CBLatch_F_2       <= "0000";
  elsif MDLE = '1' then
     CBLatch_F_2(6 downto 4) <= CB_In(6 downto 4);
     CBLatch_F_2(7)          <= DParIO_In;
  end if;
 end process;

CBLatch_F <= CBLatch_F_2 & CBLatch_F_1;


  BullEn_PROC: process(Mem_Test, ByteSel)
  begin
   BullEn <= not (ByteSel(0) and ByteSel(1) and ByteSel(2) and ByteSel(3)) or Mem_Test;
  end process;

  IUChk_Out_Gen: process (IUDataLatch_F, ChkGen_Data, BullEn, CB_bull)
   Variable IUGen_Chk  : std_logic_vector(7 downto 0);

  begin
    IUGen_Chk(6 downto 0) := ChkGen (IUDataLatch_F);
    IUGen_Chk(7)          := ChkGen_Data(32);
   CB_Out_Int <= mux2 (BullEn, IUGen_Chk, CB_bull);
  end process;
```

A problem with the dataflow method is also the low abstraction level. The functionality is coded with simple constructs typically consisting of multiplexers, bit-wise operators and conditional assignments (if-then-else). The overall algorithm (e.g. non-restoring division) might be very difficult to recognize and debug.

Yet an other issue is simulation time: the assignment of a signal takes approximately 100 times longer than assigning a variable in a VHDL process. This is because the various signal attributes must be updated, and the driving event added to the event qeueue. With many concurrent statments and processes, a larger proportion of the simulator time will be spent managing signals and scheduling of processes and concurrent statements.

5.3 The goals and means of the 'two-process' design method

To overcome the limitations of the dataflow design style, a new 'two-process' coding method is proposed. The method is applicable to any synchrounous single-clock design, which represents the majority of all designs. The goal of the two-process method is to:

- Provide uniform algorithm encoding
- Increase abstraction level
- Improve readability
- Clearly identify sequential logic
- Simplify debugging
- Improve simulation speed
- Provide one model for both synthesis and simulation

The above goals are reached with suprisingly simple means:

- Using record types in all port and signal declarations
- Only using two processes per entity
- Using high-level sequential statements to code the algorithm

The following section will outline how the two-process method works and how it compares with the traditional dataflow method.

5.4 Using two processes per entity

The biggest difference between a program in VHDL and standard programming language such C, is that VHDL allows concurrent statements and processes that are scheduled for execution by events rather than in then order they are written. This reflects indeed the dataflow behaviour of real hardware, but becomes difficult to understand and analyse when the number of concurrent statments passes some threashold (e.g. 50). On the other hand, analysing the behaviour of programs witten in sequential programming languages does not become a problem even if the program tends to grow, since there is only one thread of control and execution is done sequentially from top to bottom.

In order to improve readability and provide a uniform way of encode the algorithm of a VHDL entity, the two-process method only uses two processes per entity: one process that contains all combinational (asynchronous) logic, and one process that contains all sequential logic (registers). Using this structure, the complete algorithm can be coded in sequential (non-concurrent) statements in the combinational process while the sequential process only contains registers, i.e. the state.

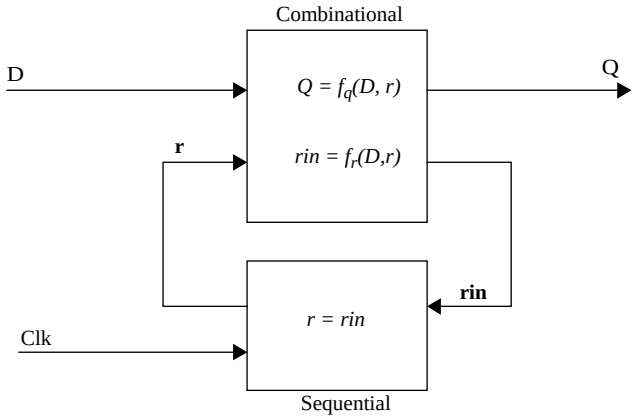

Figure 20: Generic two-process circuit

Figure 20 above shows a block diagram of a two-process entity. Inputs to the entity are denoted *D* and connected to the combinational process. The inputs to the sequential process are denoted *rin* and driven by the combinational process. In the sequential process, the inputs (*rin*) are copied to the outputs (*r*) on the clock edge,

The functionality of the combinational process can be described in two equations:

$$Q = f_q(D, r) \qquad rin = f_r(D, r)$$

Given that the sequential process only perform a latching of the state vector, the two functions are enough to express the overall functionality of the entity.

The corresponding VHDL code for an 8-bit counter could look like the following:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity count8 is
  port (
    clk   : in  std_logic;
    load  : in  std_logic;
    count : in  std_logic;
    d     : in  std_logic_vector(7 downto 0);
    q     : out std_logic_vector(7 downto 0));
end;

architecture twoproc of test is

signal r, rin : std_logic_vector(7 downto 0);

begin

  combinational : process(load, count, d, r)
  variable tmp : std_logic_vector(7 downto 0);
  begin
    if load = '1' then tmp := d;
    elsif count = '1' then tmp := r + 1;
    else tmp := r; end if;
    rin <= tmp;
    q <= r;
  end process;

  sequential : process(clk)
  begin
    if rising_edge(clk) then r <= rin; end if;
  end process;

end;
```

5.5 Using record types

The above count8 example is simple, and the limited number of ports and signals makes the code reasonably readable. However, the port interface list can for complex IP blocks consist of several hundreds of signals. Using the standard dataflow method, the signals are not grouped into more complex data types but just listed sequentially. The most common data types are scalar types and one-dimentional arrays (buses). Having a port list of several hundreds of signals makes it difficult not only to understand which signals functionally belong together, but also to add and remove signals. Each modification to the interface list has to be made at three separe locations: the entity declaration, the entity's componenent declaration, and the component instantiation (adding a port map).

By using record types to group associated signal, the port list becomes both shorter and more readable. The signals are grouped according to functionality and direction (in or out). The record types can be declared in a common global 'interface' package which is imported in each module. Alternatively, the record types can be declared together with the entity's component declaration in a 'component' package. This package is then imported into those modules where the component is used. A modification to the interface list using record types corrsponds to adding or removing an element in one of the record types. This is done only in one single place, the package where the record type is declared. Any changes to this package will automatically propagate to the component declaration and the entity's component instantiation, avoiding time-consuming and error-prone manual editing.

Similar problems arise when more registers are added. For each register, two signals have to be declared (register input and output), the register output signal has to be added to the sensitivity list of the combinational process, and an assignment statement added

to the sequential process. By grouping all signals used for registers into one record type, this becomes unnecessary. The *rin* and *r* signals becomes records, and adding register is done by simply adding a new element in the register record type definition.

Below is the count8 example using records for port and register signals. The load and count inputs are now latched before being used, and a zero flag has been added:

```
library ieee;
use ieee.std_logic_1164.all;

package count8_comp is        -- component declaration package
  type count8_in_type is record
    load  : std_logic;
    count : std_logic;
    din   : std_logic_vector(7 downto 0);
  end;

  type count8_out_type is record
    dout  : std_logic_vector(7 downto 0);
    zero  : std_logic;
  end;

  component count8
  port (
    clk   : in  std_logic;
    d     : in  count8_in_type;
    q     : out count8_out_type);
  end component;
end package;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.count8_comp.all;

entity count8 is
  port (
    clk   : in  std_logic;
    d     : in  count8_in_type;
    q     : out count8_out_type);
end;

architecture twoproc of count8 is
  type reg_type is record
    load  : std_logic;
    count : std_logic;
    zero  : std_logic;
    cval  : std_logic_vector(7 downto 0);
  end;
  signal r, rin : reg_type;
begin

  comb : process(d, r)                        -- combinational process
  variable v : reg_type;
  begin
    v := r;                                   -- default assignment
    v.load := d.load; v.count := d.count;     -- overriding assignments
    v.zero := '0';

    if r.count = '1' then v.cval := r.val + 1; end if;   -- module algorithm
    if r.load = '1' then v.cval := d.data; end if;
    if v.cval = "00000000" then v.zero := '1'; end if;

    rin <= v;                                 -- drive register inputs
    q.dout <= r.cval; q.zero <= r.zero;       -- drive module outputs
  end process;

  regs : process(clk)                         -- sequential process
  begin
    if rising_edge(clk) then r <= rin; end if;
  end process;
end;
```

Note the usage of the variable v in the combinational process. The variable is of the register record type, and assigned in the begining of the process with the value or *r*, i.e. the current register values. At the end of the process, the register inputs *rin* are assigned with v. This means that those elements of v which are not assigned during the execution of the process will maintain their values, i.e. the register value will not change.

A large benefit with using record types for the register signals *rin* and *r*, is that elements can added or removed without requiring any other modifications to the code. The sensitivity list of the combinational process does not have to be modified, and neither the assignment of *r* <= *rin* in the sequential process. This is because the operation is performed on the record as whole, regardless of how many elements it has.

In larger blocks with many registers, readability can be improved by defining separate record types for related registers. This is particularly usefull if several register of the same type are used, in which case an array type of the 'sub-register' can be defined:

```
type uart_rx_reg_type is record
  par   : std_logic;
  frame : std_logic;
  ready : std_logic;
  data  : std_logic_vector(7 downto 0);
end;

type uart_tx_reg_type is record
  par   : std_logic;
  ena   : std_logic;
  empty : std_logic;
  baud  : std_logic_vector(7 downto 0);
end;

type uart_rx_arr is array 0 to 3 of uart_rx_reg_type;
type uart_tx_arr is array 0 to 3 of uart_tx_reg_type;

type reg_type is record
  rxregs  : uart_rx_arr;
  txregs  : uart_tx_arr;
end;

signal r, rin : reg_type;
```

5.6 Clock and reset

In the examples above, the clock signal has not been included in the record types used for ports. The clock is typically routed from an input pad and through the complete hierarchy of modules. In a synchronous single-clock design, the clock may not be skewed or the function cannot be guaranteed. If the clock was included in a record type, the assignment to the record field would create a delta delay, skewing that part of the clock tree. An other (less 'noble') reason not to add the clock to a record type is that many CAD tools used for clock-tree generation and timing analysis cannot handle a clock signal that is part of a bus!

Also the reset signal has been left out from the record types, much for the same reasons as the clock signal. This reasoning is valid if the reset is asynchronous, it must then be treated as a clock both during routing and timing analysis. A synchronous reset signal can be added to the record types since it behaves like any other non-clock input signal.

The two-process methodology can handle both synchronous and asynchronous reset, but using different coding style. A synchronous reset is treated as any other input signal and used in the combinational process. By placing the reset assignment last in the process, it will have precedense before any other statments:

```
entity count8 is
  port (
    clk   : in  std_logic;
    reset : in  std_logic;
    d     : in  count8_in_type;
    q     : out count8_out_type);
end;
```

```
architecture twoproc of count8 is
.
.
begin
.
.
  comb : process(reset, d, r)                -- combinational process
  variable v : reg_type;
  begin
    v := r;                                  -- default assignment
    v.load := d.load; v.count := d.count;    -- overriding assignments
    v.zero := '0';

    if r.count = '1' then v.cval := r.val + 1; end if;   -- module algorithm
    if r.load = '1' then v.cval := d.data; end if;
    if v.cval = "00000000" then v.zero := '1'; end if;

    if reset = '0' then                                  -- reset condition
      v.cval := (others => '0'); v.zero := '0';
    end if;

    rin <= v;                                -- drive register inputs
    q.dout <= r.cval; q.zero <= r.zero;      -- drive module outputs
  end process;
```

An asynchronous reset must be connected to the sequential process, since it will affect the state (registers) regardless of the clock:

```
  regs : process(reset, clk)                 -- sequential process
  begin
    if reset = '0' then
      r.cval <= (others => '0'); r.zero <= '0';
    elsif rising_edge(clk) then r <= rin; end if;
  end process;
```

The reset signal must be added to the sequential process' sensitivity list since reset should occur regardless of the clock. The above coding style is fully synthesisable and will produce flip-flops with asynchronous reset with most synthesis tools. Asynchronous set is created simply by changing the reset assignment value from '0' to '1'. The polarity of the set/reset is defined in the asynchronous reset condition (if reset = '0'/'1').

5.7 Hierarchical design

Using record types for ports also simplifies hierarchichal design. The port map of instantiated components is reduced to a few record signals, thereby increasing the readability. Below is an example from the LEON2 processor, instantiating the processor pipeline, floating-point unit and caches. With the traditional dataflow method, there would be many hundreds of signals in the port maps. Using record types reduces this to a few signals per component, and significantly improves readability:

```
cpu0 : cpu_sparc port map (rst, clk, ici, ico, dci, dco, fpui, fpuo);
fpu0 : fpu_core port map (clk, fpui, fpuo);
cache0 : cache port map (rst, clk, ici, ico, dci, dco, ahbi, ahbo, ahbsi, crami, cramo);
cmem0 : cachemem port map (clk, crami, cramo);
```

5.8 Increasing the abstraction level

An important step towards a more efficient design methodology is to increase the abstraction level in the design process. Describing an adder with a '+' rather a network of AND, OR and XOR gates is much more readable and also less error-prone. The two-process method uses sequential VHDL statement to code the algorithm of a function, and allows the usage of more complex and abstract syntax then avalable for concurrent statements (in the dataflow method). Some ways of increasing the abtraction level for common digital operations is described below.

5.8.1 Ieee.numeric_std package

The ieee.numeric_std package defines many usefull arithmetic operations, and is provided free of charge by IEEE. Most simulators and synthesis tools provide built-in, optimised versions of this package which further improves performances and synthesis results. In paricular the +, - and compare operators are mapped on the best implementation style for a given target technology, and the usage of these operators will guarantee optimal design portability.

5.8.2 Loop statement

The loop statement is well suited to implement iterative algorithms, as well as priority encoding, sub-bus extraction and bus index inversion. The loop statement is supported by most synthesis tools as long as the loop range is constant. Some examples:

```
variable v1 : std_logic_vector(0 to 7);
variable first_bit : natural;

-- find first bit set
for i in v1'range loop
  if v1(i) = '1' then
    first_bit := i; exit;
  end if;
end loop;

-- reverse bus
for in 0 to 7 loop v1(i) := v2(7-i); end loop;
```

5.8.3 Multiplexing using integer conversion

Implementing multiplexers and decoders using integer conversion is a more compact and scalable alternative to a large 'case' statement. Generic multiplexers can for instance be implemented as functions:

```
-- generic multiplexer

function genmux(s,v : std_ulogic_vector) return std_ulogic is
variable res : std_ulogic_vector(v'length-1 downto 0);
variable i : integer;
begin
  res := v; i := 0;
  i := to_integer(unsigned(s));
  return(res(i));
end;

-- generic decoder

function decode(v : std_ulogic_vector) return std_ulogic_vector is
variable res : std_ulogic_vector((2**v'length)-1 downto 0);
variable i : natural;
begin
  res := (others => '0'); i := 0;
  i := to_integer(unsigned(v));
  res(i) := '1';
  return(res);
end;
```

5.8.4 State machines

Using sequential statements, a state machine can easily be implemented with a 'case' statement. Using the two-process method with a local varaible v to hold the next state, both combinational and registered outputs from the state machine is possible:

```
architecture rtl of mymodule is

  type state_type is (first, second, last);

  type reg_type is record
    state : state_type;
    drive : std_logic;
  end record;

  signal r, rin : reg_type;

begin
  comb : process(...., r)
  variable v : reg_type;
  begin

    v := r;

    case r.state is
    when first =>
      if cond0 then v.state := second; end if;
    when second =>
      if cond1 then v.state := first;
      elsif cond2 then v.state := last; end if;
    when others =>
      v.drive := '1'; v.state := first;
  end case;

  if reset = '1' then v.state := first; end if;

  modout.cdrive <= v.drive; -- combinational output
  modout.rdrive <= r.drive; -- registered output

  end process;
```

5.8.5 Sub-programs

Using sub-programs (procedures and functions) is a powerfull method to hide complexity and improve readability. Sub-programs are readily supported by synthesis tools, but may normally not contain sequential (clocked) logic. This restriction fits well with the two-process method which only contains combinational logic in the algorithm part. Tested and reusable sub-programs can be kept in a separate package and use as a IP library of small algorithms. Below is an example from the LEON3 processor pipeline:

```
----------------------------------------------------------------------
-- REGFILE STAGE
----------------------------------------------------------------------

    exception_detect(r, v.r.ctrl.trap, v.r.ctrl.tt);

    op_gen(r, rfo.data1, ex_alu_res, me_bp_res, zero32, r.r.rs1, false,
      v.e.op1, v.e.ldbp1);
    op_gen(r, rfo.data2, ex_alu_res, me_bp_res, imm_data(r, rf_icc),
      r.r.rs2, imm_select(r.r.ctrl.inst), v.e.op2, v.e.ldbp2);

    alu_op(r, v.e.op1, v.e.op2, v.e.aluop, v.e.alusel, v.e.aluadd, v.e.shcnt, v.e.smsb,
      v.e.shleft);
    v.e.alucin := cin_gen(v, r);

----------------------------------------------------------------------
-- DECODE STAGE
----------------------------------------------------------------------

    su_et_select(r, v.r.su, v.r.et);
    v.r.ctrl.wicc := write_icc(r.d.inst);
    de_cwp := cwp_select(r);
    v.r.ctrl.cwp := ncwp_gen(r.d.inst, de_cwp);
    cwp_ctrl(r,v.r.ctrl.cwp, v.r.ctrl.wcwp, v.r.wovf, v.r.wunf);
    v.r.ctrl.rd := regaddr(v.r.ctrl.cwp, r.d.inst(29 downto 25));
    de_rs1 := rs1_gen(r);
```

5.9 Dataflow vs. two-process comparison

To illustrate the usefullness of the two-process method, a comparison of common development tasks has been made with the standard dataflow design style:

	Two-process method	Dataflow coding
Adding ports	• Add field in interface record type	• Add port in entity declaration • Add port to sensitivity list (input) • Add port in component declaration • Add signal to port map of component • Add definition of signal in parent
Adding registers	• Add field in register record type	• Add two signal declaration (d & q) • Add q-signal in sensitivity list • Add driving signal in comb. process • Add driving statement in seq. process
Debugging	• Put a breakpoint on first line of combination process and step forward • New signal values visible in local variable v	• Analyze how the signal(s) of interest are generated • Put a breakpoint on each process or concurrent statment in the path • New signal value not immediately visible
Tracing	• Trace the r-signal (state) • Automatic propagation of added or deleted record elemenets	• Find all signals that are used to implement registers • Trace all found signals • Re-iterate after each added or deleted signal
	•	•

Table 16: Dataflow vs. two-process comparison

From the table, it can be seen that common development tasks are done with less editing or manual procedures, thereby improving efficiency and reducing coding errors.

5.10 Summary and conclusions

The presented two-process method is a way of producing structured and readable VHDL code, suitable for efficient simulation and synthesis. By defining a common coding style, the algorithm can be easily identified and the code analysed and maintained also by other engineers than the main designer. Using sequential VHDL statements to code the algorithm also allows the use of complex statements and a higher abtraction level. Debugging and analysis is simplified due to the serial execution of statements, rather than the parallel flow used in dataflow coding.