# An Innovative Application Of VHDL To Model The Interface Between A Data Path and A Controller.

Egbert Molenkamp
Department of Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
the Netherlands
email: molenkam@cs.utwente.nl

**Abstract.**

Since data path and controller are often written as separate subsystems a technique is needed that makes the VHDL code readable (self-documented), easy to change, and less error prone.

At first glance there seems to be no problem at all. But the real problem occurs when describing real systems, like CISC processors, with many control signals from controller to data path. Furthermore this number of control signals often changes regularly during the design process.

This paper describes a technique that will be appropriate if the number of control lines changes during the design process. Furthermore the requirements mentioned earlier are preserved.

## 1. Introduction.

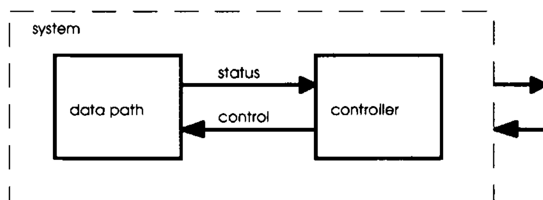


**Figure 1**. Typical division of a digital system.

Figure 1 shows a typical division of a digital system in a data path and a controller subsystem. In VHDL these two subsystems are often described as two separate entities. The whole system is described as a structural VHDL description in which both components are instantiated.

For a normal system, e.g. a CISC processor, the number of *control* signals are large, about hundred control lines. Whereas the number of *status* signals from data path to controller is rather small.

The real problem is how to describe this in VHDL with the following requirements in mind:

1. Easy to read (self-documented).
2. Not error prone.
3. Easy to change.
4. Easy to write.

## 2. Problem description.

In essence it is not difficult to produce such a VHDL description. But it is very difficult to conform with all four requirements mentioned in chapter 1. Lets concentrate on the control signals only. Suppose we have about hundred control signals, with logical names 'r1_eo' (register r1 enable output), 'r2_eo', ..., 'r1_ei', 'r2_ei', etc.

There are two straightforward ways to describe this in VHDL:

1. Separate interface declarations for each logical name.

2. Use of an array that contains all control signals.

### 2.1. Separate interface declarations for each logical name.

Figure 2 shows a global structure of a VHDL description in which the logical names of the control signals are used to improve readability. However, there are a number of disadvantages:

1. The port declaration is very large due to the amount of control lines.
2. The architectural description of the controller is large, since in each state all control signals should have a new value.
3. Addition and removal of control lines is a nasty task.

The last disadvantage can be explained as follows. Assume you want to add a control line. This means you have to change:

a. The entity description of the data path and the controller. Errors made are detected during analysis.

b. In the architecture description of the controller you have to add at each state the new control signal. An error is only detected in the simulation phase. Since if a control line is not mentioned at a state its value is unchanged, which is still correct VHDL.

c. In the structural description of the system you have to add an additional signal declaration, and the port map should be changed.

From the previous it is clear that this is not a fine solution.

```
entity datapath is
   port ( r1_eo :   in bit;
          r2_eo :   in bit;
          ..
          r1_ei :   in bit;
          r2_ei :   in bit
          );
architecture behavior of datapath is
 ..
end behavior;

entity controller is
   port ( r1_eo :   out bit;
          r2_eo :   out bit;
          ..
          r1_ei :   out bit;
          r2_ei :   out bit
          );

architecture behavior of controller is
   ..
begin
   process
      ..
   begin
      case state is
         when idle =>   r1_eo <= '1';
                        r2_eo <= '0';
                        ....
end behavior;

entity system is
   ..
end system;

architecture structure of system is
   component datapath
      ..
   component controller
      ..
   signal r1_eo, r2_eo, ..., r1_ei, r2_ei, ..
begin
   dp : datapath port map (....);
   ct : controller port map (....);
end structure;
```

**Figure 2.** Global structure of the VHDL description with separate interface declarations for each control line.

### 2.2. Use of an array that contains the control signals.

The description in figure 3 is more compact than that of figure 2 due to the fact that no separate enumeration is used for the control signals neither in the entity description of the data path and the controller nor in the structural description of the whole system.

```
entity datapath is
   port (control : in control_bus;
          ..
          );
architecture behavior of datapath is
   ..
end behavior;

entity controller is
   port (control : out control_bus;
          ..
          );

architecture behavior of controller is
   ..
begin
   process
      ..
   begin
      case state is
         when idle =>
               control <= "100110110011.."
         ..
end behavior;

entity system is
   ..
end system;

architecture structure of system is
   component datapath
      ..
   component controller
      ..
   signal control : control_bus;
begin
   dp : datapath port map (control, ..);
   ct : controller port map (control, ..);
end structure;
```

**Figure 3.** Global structure of the VHDL description with an array that contains the control signals.

Each control signal now has its own position in the control_bus (an array) which is rather large due to amount of control signals. Therefore the following problems arise:

1. It is error prone. Is a '1' placed on the correct position in the array?
2. Not suitable for documentation. In the controller description signal assignments like:
   control <= "0100010000010001001110 00..";
   are found, but its meaning?
3. Data path description not readable, e.g.

```
process
begin
   wait until clk='1' and clk'event;
   if control (13)='1'
      then r1 <= input_r1;
   end if;
   ...
```

   But is position '13' the *input enable* of the register. It is not self-documented. The latter could be improved by adding aliases for the control signals, like:

```
alias r1_ei : bit is control(13);
```

   But this also increases the length of the code.

### 3. A solution: use only the names of the control signals gathered in an array.

It is possible to combine the advantages of the previous solutions. The usage of an array for the control signals is nice. But filling it with ones and zeros by hand is repulsive. Furthermore the exact position of a control signal in the array is not important, it should only be fixed in the whole description.

Let us first restrict to the case that control signals are only '1' or '0'. Afterwards this is extended with a don't care, which is often a suitable value for control signals.

In our solution the designer only has to write a package *control_names* that contains one type *control_signals* which is an enumeration of the logical names used.

```
package control_names is
   type control_signals is
      (r1_eo, r2_eo, ..., r1_ei, r2_ei, ..);
end control_names;
```

Furthermore there is a predefined package *control_types*. The package declaration is (for a more complete description see appendix 1):

```
use work.control_names.all;
package control_types is
   type control_bit3 is ( '1', '0', '-' );
                                    -- '-' is don't care
```

```
    type control_signals_vector is array
        (natural range <>) of control_signals;
    type control_bus is array
        (control_signals) of control_bit3;
    -- The functions 'ctrl1' makes the listed
    -- control names '1' in the control_bus,
    -- the not listed control names are '0';
    function ctrl1 (inp1 : control_signals_vector)
            return control_bus;
    function ctrl1 (inp1 : control_signals)
            return control_bus;
    function ctrl1 return control_bus;
        ....
end control_types;
```

In this package declaration the *control_bus* is an array of *control_signals*. Therefore the position of a control signal in the *control_bus* is not a number but its own name!

Assume in the data path we have the following entity declaration:

```
use work.control_names.all;
use work.control_types.all;
entity datapath is
    port (control : in control_bus; .. );
```

Than in the architectural description you can write readable code, like:

```
process
begin
    wait until clk='1' and clk'event;
    if control (r1_ei)='1'
        then r1 <= input_r1;
    end if;
    ...
```

But also the controller is readable, and easy to write. The architectural description of the processor could have the following form:

```
process
    ..
begin
    case state is
        when idle => control <= ctrl1;
        when init => control <= ctrl1(r1_ei);
        when add =>
                control <= ctrl1 (r1_ei & r3_eo);
        ...
```

The control lines that should be '1' are listed in the function call of *ctrl1*. Three cases are possible, all shown in the example above. For these three cases overloaded functions are defined in the package *control_types*:

1. All control signals are '0'.
   ( control <= ctrl1; )
2. Exactly one control signal is '1'.
   ( control <= ctrl1(r1_ei); )
3. two or more control signals are '1'.
   ( control <= ctrl1 (r1_ei & r3_eo); )

**3.1. Evaluation of the use of logical names only.**

This technique has a lot of advantages:

1. The code is readable, since the logical names of the control signals are used. Hence, also less error prone.
2. Only at one place you have to enumerate the used control_signals (in the package control_names).
3. You can easily add a control signal by adding it to the enumeration type *control_signals*. The position is not important. After the number of control signals is changed. You don't need to change the interface declaration of entity and component instantiations. Consistency is guaranteed.
4. The VHDL descriptions are easy to write.

Appendix 2 gives a complete example using this technique for a multiplier.

**3.2. General method.**

In general often a control signal has a value that is don't care. In the previous solution a control signal is '1' if it is input of the function *ctrl1* listed, and otherwise '0'.

In the complete package there are also function available that take two arguments:

1. ctrl1_0
   The first parameter enumerates the logical names of the control signals that should be '1', and the second parameter enumerates the '0'. The not mentioned control lines are '-' (don't care).

2. ctrl1_d
   The first parameter enumerates the logical names of the control signals that should be '1', and the second parameter enumerates the don't cares. The not mentioned control lines are '0'.

3. etc.

In appendix 1 a more complete description of the predefined package is given.

## 4. Conclusion.

A new technique to describe the interface between data path and controller in VHDL is discussed. This technique is based on the use of only the names of control lines in the interface description. An array in which all the names of the control lines are gathered supports this technique.

Experiences with this new technique in our course *Digital System Design* affirm that it is less error prone and produces code that is better readable and more flexible.

**Appendix 1. A description of the packages.**

```
package control_names is
   type control_signals is
      (........   user defined enumeration
               of the logical names ......);
end control_names;
```

---

```
use work.control_names.all;
package control_types is

------------------------------------------------------------
-- This package supports the use of names in
-- stead of an index in a vector. This package can
-- be used for the control signals between the
-- datapath and the controller. The type of this
-- signal is CONTROL_BUS. The used control
-- names are to be defined in de package
-- CONTROL_NAMES.  A control_type has
-- three levels:
-- '0', '1' and '-'. The last is used as don't care.
-- example:
--   CTRL1(enable1 & ga)
--     the control signals enable1 and ga are '1'
--     and the others are '0';
--   CTRL1;
--     all control signals are '0'
--
-- THIS PACKAGE SHOULD NOT BE TO BE
-- CHANGED, only compilation is necessary
-- after package control_names.
------------------------------------------------------------

type control_bit3 is ( '1', '0', '-' );  -- '-' is don't
care
type control_bus is array (control_signals) of
      control_bit3;
type control_signals_vector is array
      (natural range <>) of control_signals;

-- The functions 'ctrl1' makes the listed control
-- names '1' in the control_bus, the not listed
-- control names are '0';
function ctrl1 (inp1 : control_signals_vector)
      return control_bus;
function ctrl1 (inp1 : control_signals)
      return control_bus;
function ctrl1 return control_bus;

-- The functions 'ctrl1_0' makes the listed control
-- names in inp1 '1', and the listed control names
-- in inp0 '0', the not listed control names are '-';
```

```
function ctrl1_0
        (inp1, inp0 : control_signals_vector)
        return control_bus;
function ctrl1_0
        (inp1 : control_signals;
        inp0 : control_signals_vector)
        return control_bus;
function ctrl1_0
        (inp1 : control_signals_vector;
        inp0 : control_signals)
        return control_bus;

-- The functions 'ctrl1_d' makes the listed control
-- names in inp1 '1', and the listed control names
-- in inpd '-', the not listed control names are '0';
function ctrl1_d
        (inp1,inpd : control_signals_vector)
        return control_bus;
function ctrl1_d
        (inp1 : control_signals;
        inpd : control_signals_vector)
        return control_bus;
function ctrl1_d
        (inp1 : control_signals_vector;
        inpd : control_signals)
        return control_bus;

   etc.
end control_types;
```

---

```
package body control_types is

function ctrl1 (inp1 : control_signals_vector)
        return control_bus is
   variable res : control_bus := (others => '0');
begin
   for i in inp1'range loop
      res(inp1(i)):='1';
   end loop;
   return res;
end ctrl1;

function ctrl1 (inp1 : control_signals)
        return control_bus is
   variable res : control_bus := (others => '0');
begin
   res(inp1):='1';
   return res;
end ctrl1;

function ctrl1 return control_bus is
   variable res : control_bus := (others => '0');
begin
   return res;
end ctrl1;

function ctrl1_0
        (inp1,inp0 : control_signals_vector)
        return control_bus is
   variable res : control_bus := (others => '-');
begin
   for i in inp1'range loop
      res(inp1(i)):='1';
   end loop;
   for i in inp0'range loop
      res(inp0(i)):='0';
   end loop;
   return res;
end ctrl1_0;

function ctrl1_0
        (inp1 : control_signals;
        inp0 : control_signals_vector)
        return control_bus is
   variable res : control_bus := (others => '-');
begin
   res(inp1):='1';
   for i in inp0'range loop
      res(inp0(i)):='0';
   end loop;
end ctrl1_0;

function ctrl1_0
        (inp1 : control_signals_vector;
        inp0 : control_signals)
        return control_bus is
   variable res : control_bus := (others => '-');
begin
   for i in inp1'range
      loop res(inp1(i)):='1';
   end loop;
   res(inp0):='0';
end ctrl1_0;

etc.
end control_types;
```

## Appendix 2: data path and controller for a multiplier.

This appendix gives as an example a complete description of a data path and a controller for a multiplier based on the shift/add algorithm for unsigned magnitude operands. A scheme of the datapath is shown in figure A2.1. In this figure a register is drawn as a double underlined box. The other components are all combinational. The boxes are operations like addition. The circles are gates with control signals, e.g. the input of register r1, i.e. ir1, is equal to a part of the sum, if control signal shiftadd is '1', or is equal to operand op1, if control signal init is '1'. If both control signals are '0' or '1' the input of register r1 is don't care.

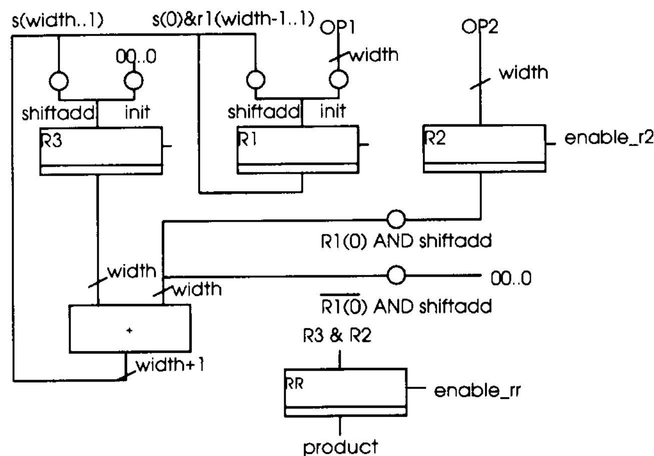


Figure A2.1 Data path of a multiplier

```
package control_names entity
  type control_signals is (init, shiftadd,
    enable_r1, enable_r2, enable_r3, enable_rr);
end control_names;

entity multiplier is
  generic (width:   natural:=4);
  port ( op1,op2 :  in bit_vector
                    (width-1 downto 0);
       product :    out bit_vector
                    (2*width - 1 downto 0));
  -- multiplier for unsigned magnitude operands
end multiplier;

use work.control_names.all;
use work.control_types.all;
entity datapath is
  generic ( width :  natural := 4);
  port ( op1,op2 :  bit_vector
                    (width-1 downto 0);
      control :     control_bus;
      clk :         bit;
      product :     out bit_vector
                    (2*width-1 downto 0));
end datapath;

use utility.math.addum;
architecture rtl of datapath is
  signal r1,r2,r3,ir1,ir2,ir3,a,b : bit_vector
            (width-1 downto 0);
  signal s: bit_vector(width downto 0);
  signal rr,irr: bit_vector(2*width-1 downto 0);
  constant zero: bit_vector(width-1 downto 0) :=
            (others => '0');
  constant dontcare: bit_vector
            (width-1 downto 0) := (others => '0');
begin
  ir1 <= op1 when control(init)='1' else
         s(0) & r1(width-1 downto 1)
            when control(shiftadd)='1' else
         dontcare;
  ir2 <= op2 when control(init)='1' else
         dontcare;
  ir3 <= zero when control(init)='1' else
         s(width downto 1)
            when control(shiftadd)='1' else
         dontcare;
  irr <= r3 & r1;
  a <= r3;
  b <= r2 when
          control(shiftadd)='1' and r1(0)='1' else
       zero when
          control(shiftadd)='1' and r1(0)='0' else
       dontcare;
  s <= addum(a,b);
  product <= rr;
  registers:process
  begin
    wait until clk='1';
    if control(enable_r1)='1' then r1<=ir1; end if;
    if control(enable_r2)='1' then r2<=ir2; end if;
    if control(enable_r3)='1' then r3<=ir3; end if;
    if control(enable_rr)='1' then rr<=irr; end if;
    end process;
end rtl;

use work.control_names.all;
use work.control_types.all;
entity controller is
```

```
 generic (width :  natural := 4);
 port (   control :  out control_bus;
          clk :      bit);
end controller;

architecture behavior of controller is
begin
 cntrl:process
 begin
  control <= ctrl1 (enable_rr & enable_r3 &
                  enable_r2 & enable_r1 & init);
  wait until clk='1';
  for i in width-1 downto 0 loop
   control<=
      ctrl1(enable_r3 & enable_r1 & shiftadd);
   wait until clk='1';
  end loop;
 end process;
end behavior;

use work.control_names.all;
use work.control_types.all;
architecture structure of multiplier is
 component datapath
 generic ( width :  natural := 4);
  port ( op1,op2 : bit_vector
                   (width-1 downto 0);
         control :  control_bus;
         clk :      bit;
         product :  out bit_vector
                   (2*width-1 downto 0));
 end component ;
 component controller
  generic (width : natural := 4);
  port ( control :  out control_bus;
         clk :      bit);
 end component ;
 signal clk : bit;
 signal control : control_bus;
begin
 dp:datapath
  generic map (width)
  port map (op1,op2,control,clk,product);
 bs:controller
  generic map (width)
  port map (control,clk);
  clk <= not clk after 5 ns;
end structure;
```