Design flow of a simple ALU

1.1. Introduction

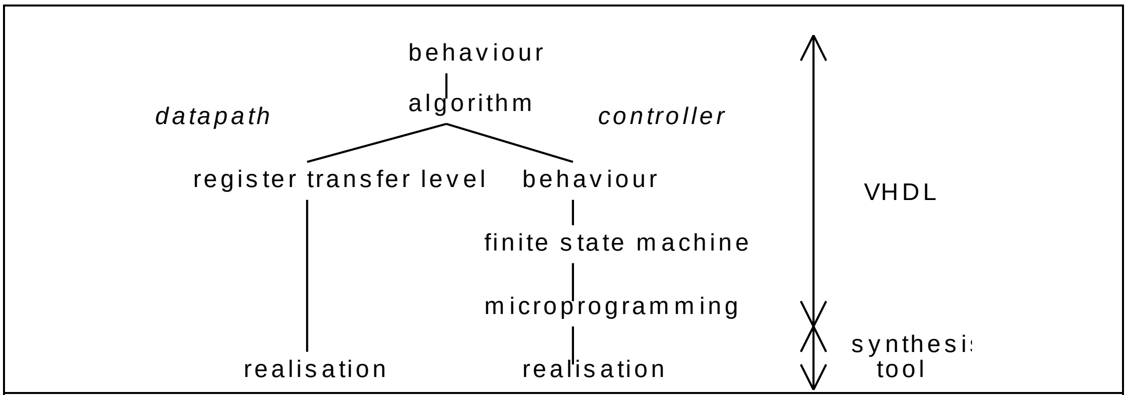


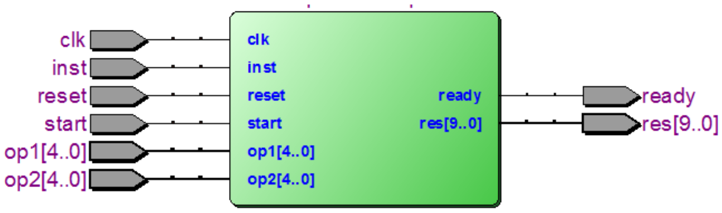
fig. 1 The design flow of a digital system

How to design a complex digital system? It is the aim of this document to illustrate the design process using a simple ALU as case. A simple example has as disadvantage that a number of design steps are not necessary due to the reduced complexity; however these design steps are taken. The design flow is shown in figure 1.[1] All design steps, except the microprogramming step, are given.
Designing is finding and consider alternative solutions to the problem with the design requirements in mind, such as speed, area, power, etc.
It is a benefit that the design steps are written in the same language. This allows comparison of a design step with the behavioural description. How this verification could be performed is also shown.

1 Many synthesis tools nowadays can synthesise VHDL description at a higher level of abstraction than shown in this figure.

Finally a VHDL description should be synthesised with a given technology in mind. However, what is the design level most suitable as input for a synthesiser, or even more important *how* should it be described in VHDL. Some synthesis tools can handle rather complex descriptions whereas others need a more detailed description.
In the standard VHDL environment a predefined type bit is declared, however this is for many descriptions not suitable, there is no three state support, no "unknown", no "don't care" value. To beat this problem an IEEE working group has defined the *std_logic_1164 multi-value logic system*. All VHDL environments, including the synthesis tools, support this package. In this chapter this package is used.
An additional problem is the integer representation of a *std_logic_vector*. What is the integer value of the vector "111"? Is it 7, or -1? This is not defined[2]. For this the IEEE developed a package numeric_std[3]. This package declares two vector types SIGNED and UNSIGNED. A signed vector is interpreted as a twos complement value whereas an unsigned vector is interpreted as a binary number (decimal 0 and higher).

1.2. Behavioural description of the ALU




```
A synchronous system (with a synchronous reset).
If input inst is low the operands op1 and op2 are added and otherwise they are
multiplied.
Before a new operation can be performed the output ready should be high.
A new operation starts when start goes to high. Start should remain high until output
ready goes to low.
If start is high then the operands and instruction should be stable.
If output ready is high the calculation is finished.
```

fig. 2 The informal specification of the ALU

Figure 2 gives the informal specification of the alu. This informal specification seems complete, but it is not. The representation of the operands is missing, is the reset active high or low? What is the active edge of the clock?
We will use VHDL to specify the system to design, this also allows simulation. There is another reason that the specification is in an executable language. It makes it possible to perform verification by simulation of a design against the specification automatically.

2 There is a non IEEE package *std_logic_unsigned* that is often used. In this package a std_logic_vector is interpreted as an unsigned.

3 There is a non IEEE package *std_logic_arith* with almost the same contents. The conversion functions in this package start with “conv_” whereas in the package *numeric_std* they start with “to_”.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity alu is
  generic (bw      : natural := 5);
  port (op1,op2    : in  std_logic_vector(bw-1 downto 0);
        inst,start : in  std_logic;
        reset      : in  std_logic;
        clk        : in  std_logic;
        ready      : out std_logic;
        res        : out std_logic_vector(2*bw-1 downto 0));
end alu;


architecture behaviour of alu is
  signal rdy_int : std_logic :='1';
begin
  alu:process
    constant add : std_logic := '0';
    constant mul : std_logic := '1';
    constant allzero : std_logic_vector(2*bw-1 downto 0) := (others => '0');
    variable op1i,op2i : unsigned (bw-1 downto 0);
  begin
    wait until rising_edge(clk);
    if reset='1' then
      rdy_int <='1'; res <= allzero;
    else
      if start='1' then
        rdy_int<='0', '1' after 50 ns; -- delay is used to simulate protocol
        op1i := unsigned(op1);
        op2i := unsigned(op2);
        case inst is
          when add    => res <= std_logic_vector(resize (('0'&op1i + op2i),2*bw));
          when others => res <= std_logic_vector(op1i*op2i);
        end case;
       end if;
     end if;
  end process alu;

  ready <= rdy_int;

  -- check I/O timing
  check_stable_inputs:process
  begin
    wait until rising_edge(clk);
    if start='1' then
      assert inst'stable report "input INST not stable" severity warning;
      assert op1'stable  report "input OP1 not stable"  severity warning;
      assert op2'stable  report "input OP2 not stable"  severity warning;
    end if;
  end process;

  transition_start_high2low:process
    variable prv_start : std_logic :='0';
  begin
    wait until rising_edge(clk);
    if prv_start='1' and start='0' then
      assert rdy_int='0' report "incorrect transition start from high to low; rdy_int
is not '0'" severity warning;
    end if;
    prv_start:=start;
  end process;

  --transition of start from high to low only allowed when ready is low
  --PSL default clock is rising_edge(clk);
  --PSL start_high2low_readylow:
  --        assert always ( {start='1'; start='0'}|-> {rdy_int='0'} );


end behaviour;
```

fig. 3 The executable specification of the alu

An executable specification of the alu is given in figure 3. The behavioural description contains a section “check I/O timing”. This section is not complete, but only illustatres this issue.
A part of the I/O relation is that after *start* goes to high it should remain high until *ready* is low. The process *transition_start_high2low* verifies this. In a similar way other I/O timing violations with respect to *start* and *ready* can be verified.
If *start* is high the data inputs should be stable in the region of the rising edge of the clock. The timing details setup and hold times will depend on technology (after the design is realized).

This behavioural description suggests that both operations (addition and multiplication) require the same number of clock cycles. Hence, the output *ready* is not really required since it is known beforehand when it is ready. However we assume that in the realization both operations may use a different number of clock cycles (“assume”: the designer is informed about this)

Assertions in VHDL can become complicated. An alternative is the use of PSL (Property Specification Language). The PSL standard is also included in the VHDL 1076-2008 standard. The comment line that start with “`--PSL`” is a PSL statement and as such it is recognized by a tool that supports PSL.

```
  --transition of start from high to low only allowed when ready is low
  --PSL default clock is rising_edge(clk);
  --PSL start_high2low_readylow:
  --          assert always ( {start='1'; start='0'}|-> {rdy_int='0'} );
```

This assertion verifies that if the sequence *start* is high followed by *start* occurs then *rdy_int* should be low when *start* is low.

1.2.1. Test environment for the behavioural description

To verify that your description is correct the design it is simulated. Exhaustive testing is in general not possible. Therefore the selection of the stimuli is important. You should use stimuli that test corner cases like the extreme values, the protocol, etc.
An example of a test environment is given in in figure 4 (it is incomplete w.r.t. the test cases). The designer has to verify that the output is correct.
At the rising edge of the clock the data is read. Therefore in the test environment the stimuli are changed on the falling edge of the clock.
Most simulation tools have a command to simulate a design until no future activities are planned (e.g. in ModelSim the command *run –all*). After all stimuli are applied the boolean *finished* is set true and the signal *clk* will not oscillate anymore.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity test_environment is
  generic (bw : natural := 5);
end test_environment;

architecture behaviour of test_environment is
  component alu is
    generic (bw      : natural := 5);
    port (op1,op2    : in  std_logic_vector(bw-1 downto 0);
          inst,start : in  std_logic;
          reset      : in  std_logic;
```

```
          clk        : in  std_logic;
          ready      : out std_logic;
          res        : out std_logic_vector(2*bw-1 downto 0));

  end component alu;
  constant addition : std_logic := '0';
  constant multiplication : std_logic := '1';
  signal reset,inst,start,ready : std_logic;
  signal clk : std_logic := '0';
  signal op1,op2 :std_logic_vector(bw-1 downto 0);
  signal res : std_logic_vector(2*bw-1 downto 0);
  signal finished : boolean := false; -- used to stop simulation
begin

  bhv: alu
    generic map (bw)
    port map (op1,op2,inst,start,reset,clk,ready,res);

  clk <= not clk after 5 ns when not finished;

  process
    -- protocol assumes that data is set (operands and instruction)
    procedure protocol (
      signal clk, ready : IN  std_logic;
      signal start : OUT std_logic) is
    begin
      assert ready='1' report "alu is busy!" severity warning;
      start<='1';
      wait until falling_edge(clk);
      lp0:loop
        wait until falling_edge(clk);
        if ready='0' then
          start<='0';
          exit;
        end if;
      end loop lp0;
      lp1:loop
        wait until falling_edge(clk);
        exit when ready='1';
      end loop lp1;
    end protocol;

    variable operand1, operand2 : integer RANGE 0 TO 2**bw-1;

  begin
    reset<='1'; start<='0';
    inst<='0'; op1<=(others=>'0'); op2<=(others=>'0');
    wait until falling_edge(clk);
    reset<='0';
    for instruction in addition to multiplication loop
      inst <= instruction;
      op1<=(others=>'0'); op2<=(others=>'0'); -- test zero operands
      protocol (clk,ready,start);
      op1<=(others=>'1'); op2<=(others=>'0'); -- test ones and zeros operands
      protocol (clk,ready,start);
      op1<=(others=>'0'); op2<=(others=>'1'); -- test zeros and ones operands
      protocol (clk,ready,start);
      op1<=(others=>'1'); op2<=(others=>'1'); -- test ones and ones operands
      protocol (clk,ready,start);
      loop_op1: for operand1 in 1 to 2**bw-1 loop  -- some other values
        op1 <= std_logic_vector(to_unsigned(operand1,bw));
        loop_op2: for operand2 in 1 to 2**bw-1 loop  -- some other values
          op2 <= std_logic_vector(to_unsigned(operand2,bw));
          protocol (clk,ready,start);
          exit loop_op2 when operand2 > 15;  -- exhaustive testing takes too long
        end loop loop_op2;
        exit loop_op1 when operand1 > 15;    -- exhaustive testing takes too long
      end loop loop_op1;
    end loop;
    assert false report "simulation finished" severity note;
    finished <= true;
    wait;
  end process;

end behaviour;
```

fig. 4 Part of an alternative architectural description of the alu

1.3. How to implement the behavioural description of the alu?

There are many ways to implement the behavior of the alu. It depends on the requirements: should it be a fast circuit? or is there only a small area available? is power consumption important? An algorithm influences these requirements. In this step alternative algorithms should be considered for the different tasks (e.g. intructions). Next the best combination of alternatives should be chosen. You should have a rough indication of the block diagram of the system.
The choosen algorithm is written in VHDL and compared with the behavioural description to detect the errors, which is commonly present in a description that is presumed to be correct. In the next chapter this comparison is discussed. The design flow assumes a top-down design approach. However, the designer should be aware of the consequences and the availability of subsystems at a lower level. Therefore a top-down design approach is more or less a 'meet in the middle' approach.
Synthesis tools can synthesise an adder. Multiplication can be a little bit more difficult especial when the operand size is large. Modern FPGA’s have a (large) number of approximately 20×20 bit combinatorial block multipliers. We assume that these block multipliers can not be used. In figure 5 an algorithm for the multiplier is given based on the shift-add algorithm. In the process the function *addshift_multiply* is declared that gives a description of the algorithm.

```
    function addshift_multiply (op1,op2 : unsigned) return unsigned is
      variable tmp : unsigned (op1'length downto 0);
      variable op1i : unsigned(op1'range);
    begin
      op1i := op1;
      tmp := (others => '0');
      for i in 1 to op1i'length loop
        if op1i(0)='1' then
          tmp := tmp + op2; --add
        end if;
        op1i := tmp(0) & op1i(op1i'length-1 downto 1); -- shift
        tmp  := '0'    & tmp(op1i'length downto 1);
      end loop;
      return tmp(op1i'length-1 downto 0) & op1i;
    end addshift_multiply;
..
    case insti is
       when add    => res <= std_logic_vector(resize (('0'&op1i + op2i),2*bw));
       when mul    => res <= std_logic_vector(addshift_multiply(op1i,op2i));
       when others => res <= allzero;
     end case;
```

fig. 5 The algorithm for multiply

1.3.1. The test environment

To verify a design step you could verify it via formal verification and check that both descriptions, the behavioural and the design step, have the same behaviour. We will use simulation. Performing a simulation in which the designer tries to verify the correctness is a bad solution, because the designer thinks that the design step is correct and therefore is probably not critical enough to detect possible errors. In large companies you often see that the verification is done by another person.
Simulation is possible but the simulation result should be best automatically compared with behavioural description and discrepancies between both descriptions should generate at least a warning.
The executable specification (not including timing) should be the same as a design step. Due to delays the circuit needs time to stabilize. Therefore the results are to be

compared after some time. In a synchronous design you can compare the results just before the next active edge of the clock, assuming that the numbers of clock cycles of the results are the same in both descriptions. In our case the latter is not true. In the specification multiplication is as fast as addition, whereas in the final result multiplication will use a number of clock cycles. Therefore we use the control signal *start* and *ready*. See figure 6 for a VHDL description. The procedure *protocol* handles the I/O including the comparison of the alu output values. Notice that in the header of this procedure the object are of mode signal! This assures that within the procedure you have access to the current values of the objects.

The test set is rather easy: extreme values are applied and some other values. In a real test environment the test data is often read from a file.

Furthermore the two component instantiations of the alu are given. This description seems to handle everything very nice: start the execution, wait for the ready signals, and finally check if errors or warnings are reported. Output *ready* is verified on the falling edge. In the real synchronous design all output will change just after the rising edge of the clock.

However, there is a problem, assume that the design step never generates a ‘1’ for *ready* then the test environment waits endlessly. In that case no warning is displayed, but the design step is not correct!

This problem can be solved using an additional VHDL process statement that checks that signal *ready* is not too many consecutive clock cycless low (i.e. alu is processing). This can also easily be verified with a PSL statement:

```
--PSL default clock is rising_edge(clk);
--PSL ready_to_long_low: assert never {not ready2[*20]};
```

If *ready2* is low for 20 consecutive clock cycles a violation is reported. More complicated checks can be written rather easily with PSL.

A separate configuration statement shows the used architectures (figure 7). This makes it possible to add other configurations using the same test environment.

```
architecture structure of test_environment is
  component alu is
    generic (bw      : natural := 5);
    port (op1,op2    : in  std_logic_vector(bw-1 downto 0);
          inst,start : in  std_logic;
          reset      : in  std_logic;
                 clk        : in  std_logic;
          ready      : out std_logic;
          res        : out std_logic_vector(2*bw-1 downto 0));
  end component alu;
  constant addition : std_logic := '0';
  constant multiplication : std_logic := '1';
  signal reset,inst,start,ready1, ready2 : std_ulogic;
  signal clk : std_logic := '0';
  signal op1,op2 :std_logic_vector(bw-1 downto 0);
  signal res1, res2 : std_logic_vector(2*bw-1 downto 0);
  signal finished : boolean := false; -- used to stop simulation
begin

  bhv: alu
    generic map (bw)
    port map (op1,op2,inst,start,reset,clk,ready1,res1);

  design: alu
    generic map (bw)
    port map (op1,op2,inst,start,reset,clk,ready2,res2);

  clk <= not clk after 5 ns when not finished;

  process
    -- protocol assumes that data is set (operands and instruction)
    procedure protocol (
      signal clk, ready1, ready2 : IN  std_logic;
      signal start : OUT std_logic;
         signal res1, res2 : IN std_logic_vector(2*bw-1 downto 0)) is
      variable rdy1zero,rdy2zero : boolean := false;
    begin
      assert (ready1='1') and (ready2='1') report "a alu is busy!"
             severity warning;
      start<='1';
      wait until falling_edge(clk);
      lp0:loop -- both ready should be zero, not necessarily at the same time.
        wait until falling_edge(clk);
        if ready1='0' then rdy1zero:=true; end if;
        if ready2='0' then rdy2zero:=true; end if;
        exit lp0 when rdy1zero and rdy2zero;
      end loop lp0;
      start<='0';
      lp1:loop  -- wait until both alus are ready
        wait until falling_edge(clk);
        exit lp1 when (ready1='1') and (ready2='1');
      end loop lp1;
      assert res1=res2 report "mismatch in results" severity warning;
    end protocol;

    variable operand1, operand2 : integer RANGE 0 TO 2**bw-1;

  begin
    reset<='1'; start<='0';
    inst<='0'; op1<=(others=>'0'); op2<=(others=>'0');
    wait until falling_edge(clk);
    reset<='0';
    for instruction in addition to multiplication loop
      inst <= instruction;
      op1<=(others=>'0'); op2<=(others=>'0'); -- test zero operands
      protocol (clk,ready1, ready2,start,res1,res2);
      op1<=(others=>'1'); op2<=(others=>'0'); -- test ones and zeros operands
      protocol (clk,ready1, ready2,start,res1,res2);
      op1<=(others=>'0'); op2<=(others=>'1'); -- test zeros and ones operands
      protocol (clk,ready1, ready2,start,res1,res2);
      op1<=(others=>'1'); op2<=(others=>'1'); -- test ones and ones operands
      protocol (clk,ready1, ready2,start,res1,res2);
      loop_op1: for operand1 in 1 to 2**bw-1 loop  -- some other values
        op1 <= std_logic_vector(to_unsigned(operand1,bw));
        loop_op2: for operand2 in 1 to 2**bw-1 loop  -- some other values
          op2 <= std_logic_vector(to_unsigned(operand2,bw));
          protocol(clk,ready1, ready2,start,res1,res2);
          exit loop_op2 when operand2 > 15;  -- exhaustive testing takes to long
        end loop loop_op2;
```

```
        exit loop_op1 when operand1 > 15;    -- exhaustive testing takes to long
      end loop loop_op1;
    end loop;
    assert false report "simulation finished" severity note;
    finished <= true;
    wait;
  end process;

end structure;
```

fig. 6 The test environment

```
configuration cnf_algorithm of test_environment is
  for structure
    for bhv:alu use entity work.alu(behaviour); end for;
    for design:alu use entity work.alu(algorithm); end for;
  end for;
end cnf_algorithm;
```

fig. 7 The configuration for the test environment for the algorithmic description.

1.4. Separation in datapath and controller

As mentioned previously it is assumed that there is already an appropriate implementation for an adder available. The algorithm for multiplication assumes the repeated use of an adder. Only one adder and some extra logic are necessary for the data manipulation, in the datapath, and in addition a controller to control the dataflow and operations in the datapath.
There are still design decisions to be taken, e.g. what should be the width of the internal operands? In this example the internal width of the datapath could be 1, 2 or 4 bits. However, the reduction of the data width will increase the complexity and the area of the controller, also this will influence the performance. In this case the data width of the internal operands is the same as the external ones. Figure 8 shows the division of the alu in the two subsystems.
In general the number of control signals from controller to datapath is large and will often change during the design process. Also the encoding of control signals should be open until the right moment. An enumeration type *control_bus* is declared with the logical names of the control signals, for readability and maintenance reasons, and implies no encoding. The only task left for the designer is to find the logical names for the control signals and put these in a package, for this simple alu:

```
PACKAGE control_names IS
  TYPE control_signals IS
     (enable_r1,enable_r2,enable_r3,init,shift_add,addition);

  -- do not change the following type declaration
  TYPE control_bus IS ARRAY (control_signals) OF std_logic;
END control_names;
```

```
library ieee;
use ieee.std_logic_1164.all;
use work.control_names.all;
architecture datapath_controller of alu is
  signal control : control_bus;
  component datapath
    generic (bw : natural := 4);
    port (op1,op2  : in  std_logic_vector(bw-1 downto 0);
          control  : in  control_bus;
          clk      : in  std_logic;
          res      : out std_logic_vector(2*bw-1 downto 0));
  end component;
  component controller
    generic (bw : natural := 4);
    port (inst     : in  std_logic;
          start    : in  std_logic;
          clk      : in  std_logic;
          reset    : in  std_ulogic;
          control  : out control_bus;
          ready    : out std_logic);
  end component;
begin
  dp:datapath
     generic map (bw)
     port map (op1,op2,control,clk,res);
  ct:controller
     generic map (bw)
     port map (inst,start,clk,reset,control,ready);
end datapath_controller;
```

fig. 8 The division in a datapath and a controller.

1.4.1. The datapath

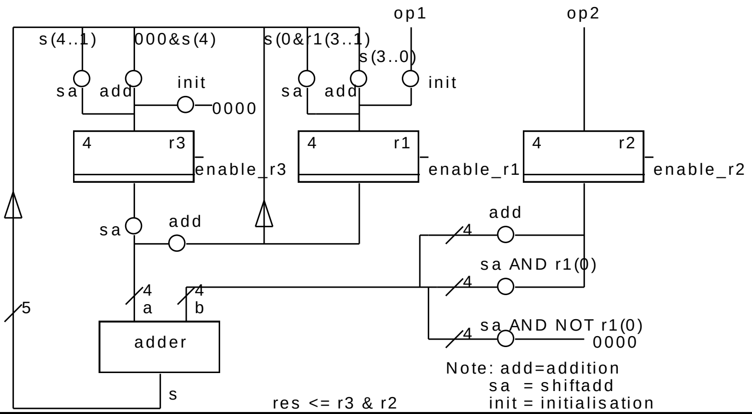
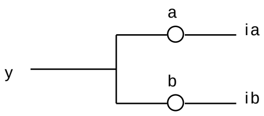


fig. 9 The block diagram of the datapath width generic value for bw is 4.[4]

---

4 Meaning of the used symbols:
- double underlined blocks
  registers (edge sensitive), possible with an enable input, the global clock line is not shown.
- other blocks
  combinational logic
- circles
  circles are gates. Only the behaviour of all connected outputs of gates can be described. If exactly one gate condition is true the output of all connected gates is equal to the input of the corresponding input of the true condition gate. If no gate condition is true, or two or more gate conditions are true the output is a don't care. In fact this is an abstraction from implementation, e.g. it could be implemented with a multiplexer.

The description of the datapath is a register transfer level description. A detailed block diagram of this description is given in figure 9. The significance of such a diagram is an informal one, it only makes the formal description easier to understand. If there is a discrepancy between the block diagram and a formal description (figure 10) of the datapath, the latter is correct.
If the control signals *init, enable_r1, and enable_r2* are '1', the operand values are read loaded in the registers at the rising edge of the clock. Next the control signals *add(ition), enable_r2, and enable_r3* are '1' an addition is performed and the result is stored in the registers *r3* and *r1*.
If the datapath contains more 'complex' elements (i.e. register file, alu, memory, ..) component instantiations can be used. This has the advantage that each design entity of a sub design can be testen separately. I personally would not use a component instantion for a component that is not 'complex', e.g. a multiplexer.



The previous schematic means that the output y is equal to the value of input *ia* when control signal *a* is '1' (and control signal *b*='0'), or it is equal to the value of input *ib* when control signal *b* is '1' (and control signal a='0'). In other cases it is don't care. There is no unique way to describe this in VHDL. In fact it depends probably on the style you like. It could be written as:

```
y   <=   dontcare WHEN dontcare_condition( control(a) & control(b)) ELSE
         ia       WHEN control(a)='1'                              ELSE
         ib       WHEN control(b)='1'                              ELSE
         dontcare;
```

The function *dontcare_condition* takes the control values as input and determines whether there is a don't care condition or not. If both control signals are '1' *dontcare_condition* results in true.
To realise the previous schematic representation one could use a multiplexer, or 'pass transistors' for the gates, and there are many more possibilities.
From a specification point of view the don't care condition check is nice. However the synthesis tool probably may generate a larger and/or slower circuit. Therefore consider using a more practical solution and don't use the dontcare_condition check:

```
y   <=   ia       WHEN control(a)='1'                              ELSE
         ib;      WHEN control(b)='1'                              ELSE
         dontcare;
```

You could add an assertion with the function *dontcare_condition* and report a warning if multiple control signals are '1'.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use work.control_names.all;
entity datapath is
  generic (bw : natural := 4);
  port (op1,op2  : in  std_logic_vector(bw-1 downto 0);
        control  : in  control_bus;
        clk      : in  std_logic;
        res      : out std_logic_vector(2*bw-1 downto 0));
end datapath;

library ieee;
use ieee.numeric_std.all;
architecture rtl of datapath is
  constant zerobw_1      : std_logic_vector(bw-2 downto 0) := (others=>'0');
  constant zerobw        : std_logic_vector(bw-1 downto 0) := (others=>'0');
  constant dontcare      : std_logic_vector(bw-1 downto 0) := (others=>'-');
  signal s               : std_logic_vector(bw downto 0);
  signal a,b,ir1,ir2,ir3 : std_logic_vector(bw-1 downto 0);
  signal r1,r2,r3        : std_logic_vector(bw-1 downto 0);

begin
  ir1 <= s(0) & r1(bw-1 downto 1) when control(shift_add)='1' else
         s(bw-1 downto 0)         when control(addition)='1'  else
         op1                      when control(init)='1'      else
         dontcare;

  ir2 <= op2;

  ir3 <= s(bw downto 1)   when control(shift_add)='1' else
         zerobw_1 & s(bw) when control(addition)='1'  else
         zerobw           when control(init)='1'      else
         dontcare;

  a   <= r3        when control(shift_add)='1' else
         r1        when control(addition)='1'  else
         dontcare;

  b   <= r2        when control(addition)='1'                else
         r2        when control(shift_add)='1' and r1(0)='1' else
         zerobw    when control(shift_add)='1' and r1(0)='0' else
         dontcare;

  s   <= std_logic_vector( ('0'&unsigned(a)) + unsigned(b));

  res <= r3 & r1;

  registers:process
  begin
    wait until rising_edge(clk);
    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;
  end process;
end rtl;
```

fig. 10 The description of the datapath.

1.4.2. The controller

The datapath description probably contains errors. However some of these errors can only be detected if the combination of datapath and controller is simulated against the behavioural description. But if there is an error detected it could be in the datapath, in the controller, or in both components. For debugging reasons as well it is very important that the controller is first described at a high level, i.e. as a behavioural description again. After simulation of this design step the controller is designed.

1.4.2.1. behavioural description of the controller

```
library ieee;
use ieee.std_logic_1164.all;
use work.control_names.all;
entity controller is
  generic (bw : natural := 2);
  port (inst     : in  std_logic;
        start    : in  std_logic;
        clk      : in  std_logic;
        reset    : in  std_logic;
        control  : out control_bus;
        ready    : out std_logic);
end controller;

architecture behaviour of controller is
  signal rdy_int : std_ulogic;
begin

  cntrl:process
    variable count : natural range 0 to bw-1;
    constant add   : std_logic := '0';
    constant mul   : std_logic := '1';
    variable insti : std_logic;
  begin
    rst: loop
      if reset='1' then
        control<=(others=>'0'); rdy_int<='1'; insti:='0';
        wait until rising_edge(clk);
      else
        lp_start:loop
          control<=(others=>'0'); rdy_int<='1';
          exit when start='1';
          wait until rising_edge(clk);
          exit rst when reset='1';
        end loop lp_start;
        insti:=inst; rdy_int<='0';
        control<=(enable_r1 | enable_r2 | enable_r3 | init => '1', others=>'0');
        wait until rising_edge(clk);
        exit rst when reset='1';
        case insti is
          when add => control<=(enable_r1 | enable_r3 | addition => '1', others=>'0');
                      rdy_int<='0';
                      wait until rising_edge(clk);
                      exit rst when reset='1';
          when others => count:=bw-1;
                      repeat:loop
                        control<=(enable_r1 | enable_r3 | shift_add => '1',
                                  others=>'0'); rdy_int<='0';
                        wait until rising_edge(clk);
                        exit rst when reset='1';
                        exit repeat when count=0;
                        count:=count-1;
                      end loop repeat;
        end case;
      end if;
    end loop rst;
  end process;

  ready <= rdy_int;

end behaviour;
```

fig. 11 The behavioural description of the controller.

The behavioural description of the controller is straightforward. However this is not the only possible behavioural description. How fast should the system respond to the reset? In this description the execution is ended rudely due to the statement `exit rst when reset='1'` after the wait statements.
For simulation the test environment is used with the configuration is shown in figure 12.

```
configuration cnf_controller_behaviour of test_environment is
  for structure
    for bhv:alu use entity work.alu(behaviour); end for;
    for design:alu use entity work.alu(datapath_controller);
      for datapath_controller
        for dp:datapath use entity work.datapath(rtl); end for;
        for ct:controller use entity work.controller(behaviour); end for;
      end for;
    end for;
  end for;
end cnf_controller_behaviour;
```

fig. 12 The configuration for the test environment for the datapath with behavioural of the controller.

1.4.2.2. finite state machine description of the controller

```
architecture fsm of controller is
  signal rdy_int : std_ulogic;
begin
  process
    type states is (idle,ini,add,mul);
    variable state : states;
    variable count : natural range 0 to bw-1;
  begin
    wait until rising_edge(clk);
    if reset='1' then
      rdy_int<='1';
      count:=0;
      state:=idle;
      control<=(others=>'0');
    else
      case state is
        when idle         => if start='1' then
                                 state:=ini;
                             end if;
        when ini          => if inst='1' then
                               state:=mul; count:=bw-1;
                             else
                               state:=add;
                             end if;
        when add          => state:=idle;
        when mul          => if count>0 then
                               count:=count-1;
                             else
                               state:=idle;
                             end if;
      end case;

      case state is
        when idle         => control<=(others=>'0'); rdy_int<='1';
        when ini          => control<=(enable_r1 | enable_r2 | enable_r3 | init=>'1',
                             others=>'0'); rdy_int<='1';
        when add          => control<=(enable_r1 | enable_r3 | addition=>'1',
                             others=>'0'); rdy_int<='0';
        when mul          => control<=(enable_r1 | enable_r3 | shift_add=>'1',
                             others=>'0'); rdy_int<='0';
      end case;
    end if;
  end process;

  ready <= rdy_int;

end fsm;
```

fig. 13 The finite state machine (fsm) description of the controller and the configuration.

In the behavioural description of the controller figure 11 the state information is not explicit. A finite state machine description is given in figure 13. Four states are used:

- idle: wait for a new operation to be performed
- initialization: the operands are read and register are initializated

- addition: operands are added,
- multiplication: this state is executed *bw* times (repeated addition and shifting).

1.5. Realisation of a VHDL description

The design entities *datapath (rtl)* (figure 10) and *controller(fsm)* (figure 13) are synthesisable, therefore the alu as a whole can be realized. The synthesis flow depends on the synthesis tool. In this document the Quartus II software of Altera is used. The input for the tooling is:

```
package pkg_control_names                       (file pkg_control_names.vhd)
entity controller                               (file controller.vhd)
architecture datapath_controller of controller  (file alu-datapath_controller.vhd)
entity datapath                                 (file datapath.vhd)
architecture rtl of datapath                    (file datapath-rtl.vhd)
entity controller                               (file controller.vhd)
architecture fsm of controller                  (file controller-fsm.vhd)
```

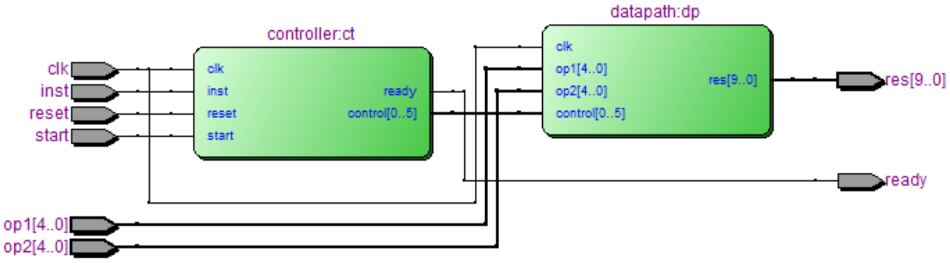
Furthermore additional constraints can be added. We want the system to operate at 100 MHz. This information is put in the synopsys design constraint file *alu.sdc*:

```
create_clock -period 10.000 -name clk [get_ports clk]
```

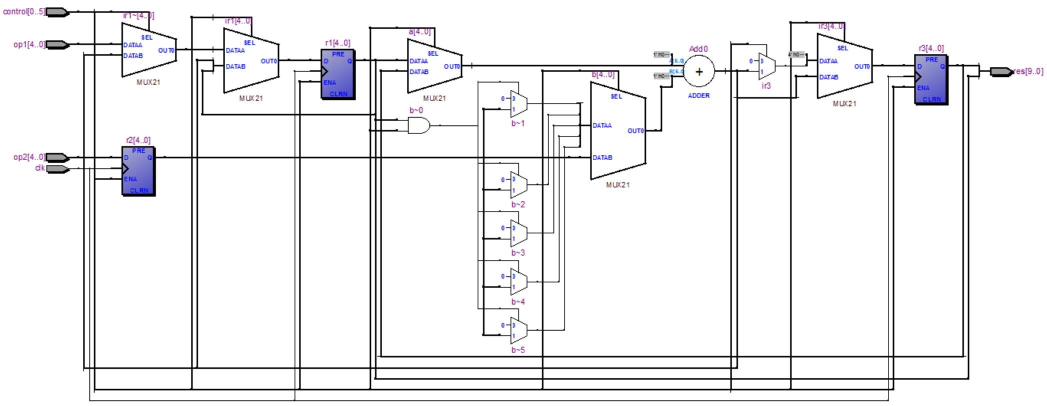
In figure 14 the realization of the alu is given. Furthermore the tool informs the user about the timing, the area and so on.
After realization the real delays are known. Most synthesis tools, also Quartus II, generate VITAL (VHDL Initiative Towards ASIC Libraries) compliant post simulation files. The names (and extensions) are not the same. For Quartus II it is:
- *.vho (VHDL output file). This is a low level structural description of the realization. You need also the technology specific packages. If you use a ModelSim-Altera version these packages are precompiled.
- *.sdo (Standard delay output file) this file contains the timing information.

In figure 15a a part if the generated output file *alu.vho* is given. Notice that the generic in the port map is replaced with constraints in the inputs *op1* and *op2* and in the output *res*. The entity name is *alu*. Since in the design flow the entity *alu* is already declared with a different port declaration the entity name in the file *alu.vho* is manually changed in *alu_realization*. Consequently also the name of the file is changed in *alu_realization.vho*.

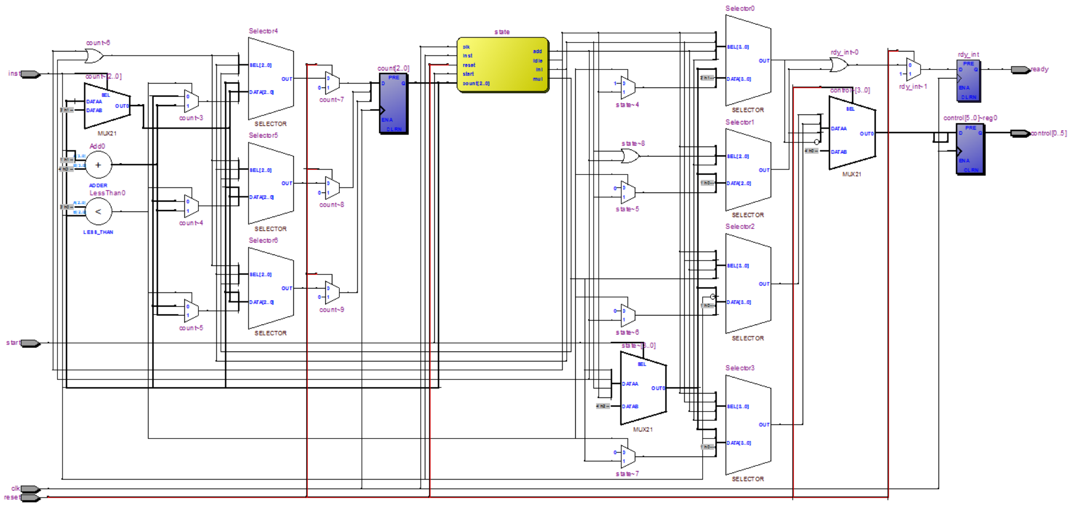
In figure 16 the whole simulation waveform is shown with the test environment (the behaviour is compared with the realization (post simulation)). In the middle part the output of the behavioural description is shown and in the bottom parts the output of the realization. Due to the delays in the post simulation you see much more activity. It can happen that the output in the realization is already high while not outputs signals are valid. It is a synchronous system, and it is known that there the clock to output delays which do not have the same values. If this is not desirable an additional state can be added to the state machine to take care of an extra clock cycle delay for the output ready.



a) RTL view of structure of alu



b) RTL view of datapath



c) RTL view of controller

fig. 14 The realization of the alu.

```
-- VENDOR "Altera"
-- PROGRAM "Quartus II 64-Bit"
-- VERSION "Version 11.0 Build 157 04/27/2011 SJ Full Version"
-- DATE "08/26/2011 09:40:42"
--
-- Device: Altera EP2C5T144C6 Package TQFP144
--
--
-- This VHDL file should be used for ModelSim-Altera (VHDL) only
--

LIBRARY CYCLONEII;
LIBRARY IEEE;
USE CYCLONEII.CYCLONEII_COMPONENTS.ALL;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY alu IS
    PORT (
        ready : OUT std_logic;
        res : OUT std_logic_vector(9 DOWNTO 0);
        op1 : IN std_logic_vector(4 DOWNTO 0);
        op2 : IN std_logic_vector(4 DOWNTO 0);
        inst : IN std_logic;
        start : IN std_logic;
        reset : IN std_logic;
        clk : IN std_logic
        );
END alu;
```

a) top of generated alu.vho output file by Quartus II. Note that manually the entity name is changed from alu to alu_realization.

```
architecture realization of test_environment is
  constant addition : std_logic := '0';
  constant multiplication : std_logic := '1';
  signal reset,inst,start,ready1, ready2 : std_logic;
  signal clk : std_logic := '0';
  signal op1,op2 :std_logic_vector(bw-1 downto 0);
  signal res1, res2 : std_logic_vector(2*bw-1 downto 0);
  signal finished : boolean := false; -- used to stop simulation
begin

  bhv: entity work.alu (behaviour)
    generic map (bw)
    port map (op1,op2,inst,start,reset,clk,ready1,res1);

  -- this instantion depends on the generated simulation output files
  -- of the synthesis tool.
  realization: entity work.alu_realization(structure)
    port map (ready=>ready2, res=> res2, op1=>op1, op2=>op2, inst=>inst,
              start=>start, reset=>reset,clk=>clk);

  clk <= not clk after 5 ns when not finished;
...
```

b) Part of the test environment with can test the realization against the behaviour

fig. 15 Part of test environment used to test the realization.

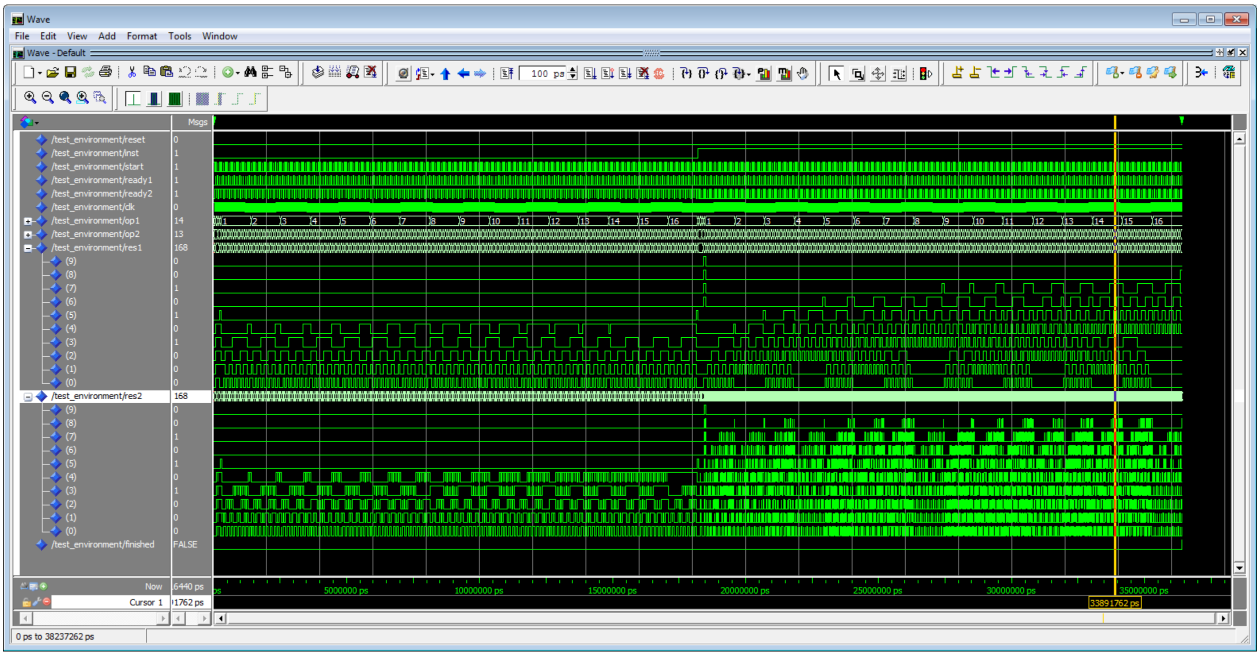


fig. 16 Part of the simulation (behaviour versus realization).

1.6. Design files

1.6.1. Design files before synthesis.

The script in file *compile_all.do* compiles all files in a correct order. Use the configurations to simulate the different design steps.

File compile_all.do

```
# packages
vcom pkg_control_names.vhd
#
# controller
vcom controller.vhd
vcom controller-behaviour.vhd
vcom controller-fsm.vhd
#
# datapath
vcom datapath.vhd
vcom datapath-rtl.vhd

# alu
vcom alu.vhd
vcom alu-behaviour.vhd
vcom alu-algorithm.vhd
vcom alu-datapath_controller.vhd
#
# test environment
vcom test_environment.vhd
vcom test_environment_behaviour.vhd
vcom test_environment_structure.vhd
#
# configuration
vcom cnf_behaviour.vhd
vcom cnf_algorithm.vhd
vcom cnf_controller_behaviour.vhd
vcom cnf_controller_fsm.vhd
```

1.6.2. Design files for post simulation

The script in file *compile_post_and simulation.do* compiles the generated output file and the test_environment.
Note: manually the entity name *alu* in the output file alu.vho is changed in *alu_realization* (also the name if the file is changed accordingly)

File compile_post_and_simulation.do

```
vcom alu_realization.vho
vcom test_environment_realization.vhd
vsim -sdftyp /realization/=alu_vhd_fast.sdo -t ps work.test_environment (realization)
```