Clock-efficient and maintainable implementation of complex state machines in VHDL

Wojciech M. Zabolotny[a]

[a]Institute of Electronic Systems, Nowowiejska 15/19, 00-665 Warszawa, Poland;

ABSTRACT

This paper presents a nonstandard approach to describe the complex state machines in VHDL to obtain both good readability of the code and efficient operation. This new approach, called “variable driven flow control in sequential process” allows to avoid loss of clock cycles when complex decisions are to be taken, and simultaneously allows to keep the structure of the code clear and easy to maintain.

A simple example has been presented, showing the idea and practical implementation of the proposed method. The code produced by the presented method is synthesizable, and the obtained parameters of resulting FPGA implementation (both speed and occupancy) are good.

Keywords: Logical synthesis, VHDL, FPGA, behavioral description

1. INTRODUCTION

When one implements a state machine in a hardware description language for FPGA implementation, it is often necessary to chose between the clearness of the code, the efficiency of the resulting implementation and the speed of operation.

The typical implementations of state machines - the one-process behavioral description and the two-process behavioral description[1] provide very good results in most typical situations, but in some more complicated cases they are not able to describe the desired functionality of the state machine in both readable and easily synthesizable way.

The next section presents a relatively simple system, which requires such a special handling. This example will be used to introduce a new method of state machine implementation.

2. PRACTICAL EXAMPLE - DATA SERIALIZER

Let us consider a simple data serializing system shown in the Fig. 1. The device receives N data bytes (D_0, D_1 ... D_{N-1}) from its parallel inputs, and sends them as a data record through the serial link. However the data record should be preceded with a configurable header which may contain three header bytes (H_A, H_B and H_C). Each header byte may be enabled or disabled independently. Some possible forms of the data record with the header are shown below:

H_A	H_B	H_C	D_0	D_1 ... D_N	–	all headers sent
	H_A	H_B	D_0	D_1 ... D_N	–	H_B omitted
		H_B	D_0	D_1 ... D_N	–	H_A and H_C omitted
			D_0	D_1 ... D_N	–	all headers omitted

The most straight-forward implementation of the above system is shown in the Fig. 2. This implementation uses the standard one-process implementation of state machine. The code is easy to understand and to modify, but it is not efficient. Omitting of any header word costs a lost clock cycle. Some simulation waveforms exposing the above effect are shown in the Fig. 3. (In all simulations the following values are assumed: $N = 10$, $H_A = $ 0x8A, $H_B = $ 0x8B, $H_C = $ 0x8C, $D_0 = $ 0x10, $D_1 = $ 0x11, ... $D_{10} = $ 0x1A).

One could be tempted to avoid the problem by running the state machine at the higher clock frequency and using the FIFO to transfer the data to the link, but such approach leads to unnecessary increase of system’s complexity and probably to increase of power consumption.

Further author information: (Send correspondence to W.M.Zabolotny)
W.M.Zabolotny: E-mail: wzab@ise.pw.edu.pl, Telephone: +48 22 6607717

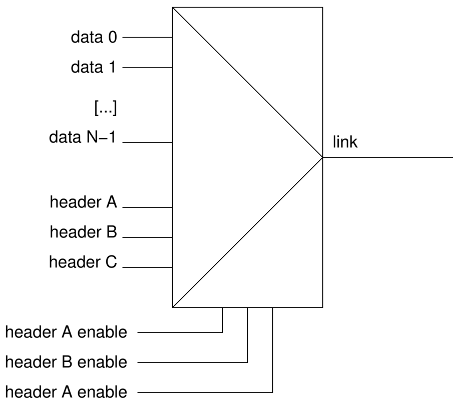

Figure 1. The block diagram of the simple data serializer used to illustrate the problem with implementation of state machines.

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

entity test1 is
  port (
    hdr_a_en   : in  std_logic;
    hdr_b_en   : in  std_logic;
    hdr_c_en   : in  std_logic;
    data       : in  T_INPUT;
    data_out   : out std_logic_vector(7 downto 0);
    data_valid : out std_logic;
    run        : in  std_logic;
    clk        : in  std_logic;
    rst        : in  std_logic);
end test1;

architecture beh1 of test1 is

  type T_STATE is (ST_IDLE, ST_HDR_A, ST_HDR_B,
                   ST_HDR_C, ST_SEND);
  signal state   : T_STATE := ST_IDLE;
  signal counter : integer range 0 to NINPUTS-1 := 0;

begin  -- beh1
  st1 : process (clk, rst)
  begin  -- process st1
    if rst = '0' then
      state      <= ST_IDLE;
      data_valid <= '0';
      data_out   <= x"00";
    elsif clk'event and clk = '1' then
      -- rising clock edge
      -- defaults
      data_valid <= '0';
      data_out   <= x"00";
      -- state definitions
      case state is
        when ST_IDLE =>
          if run = '1' then
            state <= ST_HDR_A;
          end if;
        when ST_HDR_A =>
          if hdr_a_en = '1' then
            data_out   <= HEADER_A;
            data_valid <= '1';
          end if;
          state <= ST_HDR_B;
        when ST_HDR_B =>
          if hdr_b_en = '1' then
            data_out   <= HEADER_B;
            data_valid <= '1';
          end if;
          state <= ST_HDR_C;
        when ST_HDR_C =>
          if hdr_c_en = '1' then
            data_out   <= HEADER_C;
            data_valid <= '1';
          end if;
          counter <= 0;
          state   <= ST_SEND;
        when ST_SEND =>
          data_out   <= data(counter);
          data_valid <= '1';
          if counter = NINPUTS-1 then
            state <= ST_IDLE;
          else
            counter <= counter + 1;
          end if;
        when others =>
          state <= ST_IDLE;
      end case;
    end if;
  end process st1;
end beh1;
```

Figure 2. Standard one-process implementation of the serializer's state machine

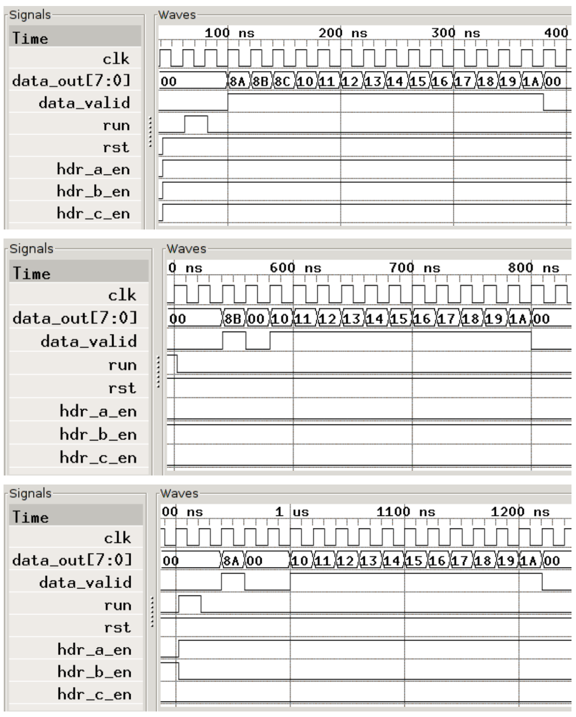

Figure 3. Functional simulation waveforms in the simplest implementation of the state machine.

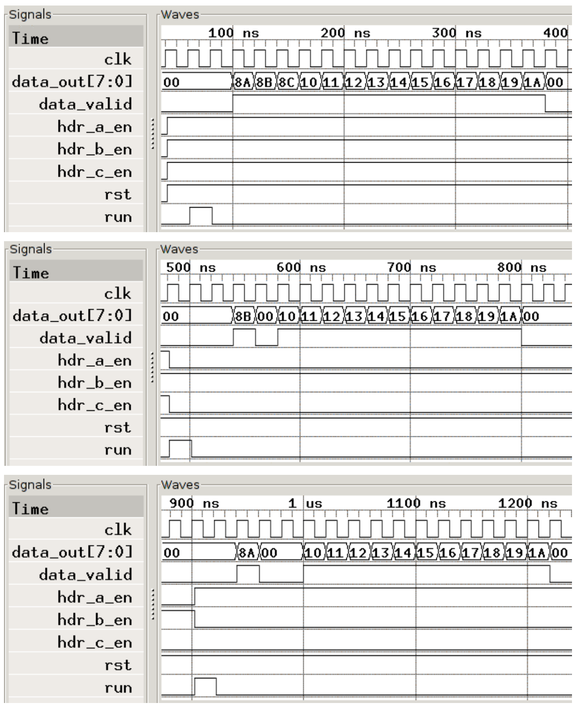

Figure 4. The functional simulation waveforms in the standard two-process implementation of state machine.

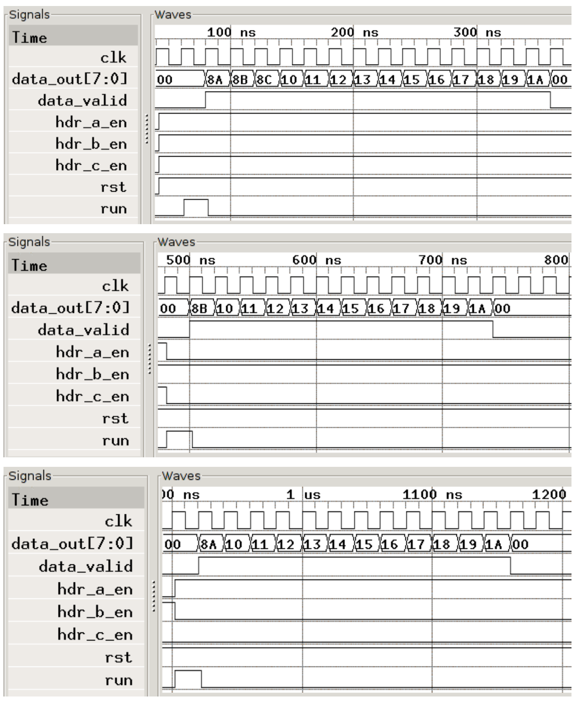

Figure 5. Functional simulation waveforms in the improved one-process implementation of the state machine.

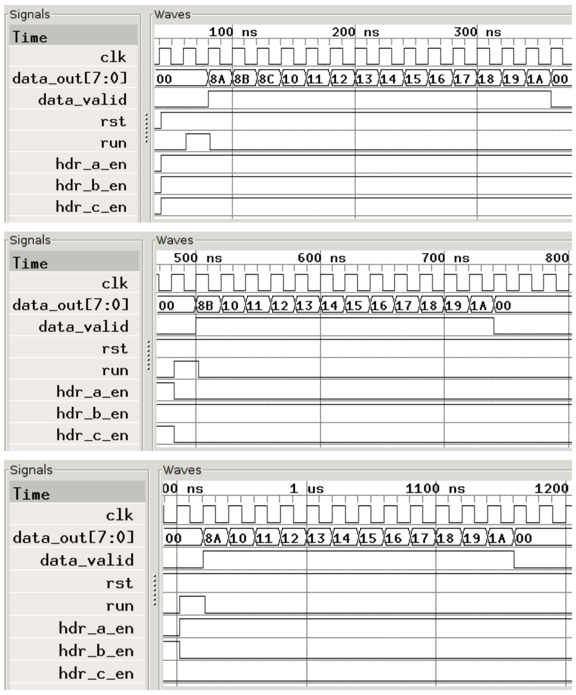

Figure 6. The functional simulation waveforms in the implementation with the “variable driven flow control in sequential process”.

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

entity test1 is
  port (
    hdr_a_en   : in  std_logic;
    hdr_b_en   : in  std_logic;
    hdr_c_en   : in  std_logic;
    data       : in  T_INPUT;
    data_out   : out std_logic_vector(7 downto 0);
    data_valid : out std_logic;
    run        : in  std_logic;
    clk        : in  std_logic;
    rst        : in  std_logic);
end test1;

architecture beh1 of test1 is

  type   T_STATE is (ST_IDLE, ST_HDR_A, ST_HDR_B,
                     ST_HDR_C, ST_SEND);
  signal state, state_next     : T_STATE  := ST_IDLE;
  signal counter,
    counter_next : integer range 0 to NINPUTS-1 := 0;
  signal run_s                 : std_logic;

begin  -- beh1

  -- sequential process
  st1s : process (clk, rst)
  begin  -- process st1s
    if rst = ’0’ then
      -- asynchronous reset (active low)
      state   <= ST_IDLE;
      counter <= 0;
      run_s   <= ’0’;
    elsif clk’event and clk = ’1’ then
      -- rising clock edge
      state   <= state_next;
      counter <= counter_next;
      run_s   <= run;
    end if;
  end process st1s;

  -- combinatorial process
  st1c : process (counter, data, hdr_a_en, hdr_b_en,
                  hdr_c_en, rst, run_s, state)
  begin  -- process st1c
    -- defaults
    data_valid   <= ’0’;
    data_out     <= x"00";
    state_next   <= state;
    counter_next <= counter;
    if rst = ’1’ then
      -- normal operation
      case state is
        when ST_IDLE =>
          counter_next <= 0;
          if run_s = ’1’ then
            state_next <= ST_HDR_A;
          end if;
        when ST_HDR_A =>
          if hdr_a_en = ’1’ then
            data_out   <= HEADER_A;
            data_valid <= ’1’;
          end if;
          state_next <= ST_HDR_B;
        when ST_HDR_B =>
          if hdr_b_en = ’1’ then
            data_out   <= HEADER_B;
            data_valid <= ’1’;
          end if;
          state_next <= ST_HDR_C;
        when ST_HDR_C =>
          if hdr_c_en = ’1’ then
            data_out   <= HEADER_C;
            data_valid <= ’1’;
          end if;
          state_next <= ST_SEND;
        when ST_SEND =>
          data_out   <= data(counter);
          data_valid <= ’1’;
          if counter = NINPUTS-1 then
            state_next <= ST_IDLE;
          else
            counter_next <= counter + 1;
          end if;
        when others => null;
      end case;
    end if;
  end process st1c;

end beh1;
```

Figure 7. Standard two-process implementation of the serializer’s state machine

The problem is associated not with the limitations of the state machines, but with the possibilities of their description in VHDL. The cause of the problem is that checking of each header enable bit and sending or omitting of the particular header word is associated with a different state. So even if the particular header word is disabled, it is necessary to use the single clock pulse to go the state associated with the next header word.

This problem may not be solved by using the standard two process implementation (Fig. 7), as this solution suffers from the same drawback (see Fig. 4).

3. ATTEMPT TO SOLVE THE PROBLEM WITH THE STANDARD METHODS

It seems, that the problem could be easily solved by checking of all header enable bits, and sending of first enabled header word in a single state. Unfortunately in this case we also need to introduce additional states for next header words, and we need to consider all possible combinations of remaining header enable bits in these states as well. The final full solution based on this idea is shown in the Fig. 8. As we can see this implementation contains a lot of “copied and pasted” code. If any corrections are to be made, they need to be performed in more than one place at the same moment, so such code is difficult to understand and to maintain.

However, as can be seen in the Fig. 5, this implementation offers good performance and no clock pulses are lost for any combination of header enable signals.

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

entity test1 is
  port (
    hdr_a_en   : in  std_logic;
    hdr_b_en   : in  std_logic;
    hdr_c_en   : in  std_logic;
    data       : in  T_INPUT;
    data_out   : out std_logic_vector(7 downto 0);
    data_valid : out std_logic;
    run        : in  std_logic;
    clk        : in  std_logic;
    rst        : in  std_logic);
end test1;

architecture beh1 of test1 is

  type   T_STATE is (ST_IDLE, ST_HDR_A, ST_HDR_B,
                     ST_HDR_C, ST_SEND);
  signal state   : T_STATE := ST_IDLE;
  signal counter : integer range 0 to NINPUTS-1 := 0;

begin  -- beh1
  st1 : process (clk, rst)
  begin  -- process st1
    if rst = '0' then
      -- asynchronous reset (active low)
      state <= ST_IDLE;
      data_valid <= '0';
      data_out <= x"00";
      counter <= 0;
    elsif clk'event and clk = '1' then
      -- rising clock edge
      -- defaults
      data_valid <= '0';
      data_out   <= x"00";
      case state is
        when ST_IDLE =>
          counter <= 0;
          if run = '1' then
            if hdr_a_en = '1' then
              data_out   <= HEADER_A;
              data_valid <= '1';
              state <= ST_HDR_B;
            elsif hdr_b_en = '1' then
              data_out   <= HEADER_B;
              data_valid <= '1';
              state <= ST_HDR_C;
            elsif hdr_c_en = '1' then
              data_out   <= HEADER_C;
              counter <= 0;
              data_valid <= '1';
              state <= ST_SEND;
            else
              data_out <= data(0);
              data_valid <= '1';
              counter <= 1;
              state <= ST_SEND;
            end if;
          end if;
        when ST_HDR_B =>
          if hdr_b_en = '1' then
            data_out   <= HEADER_B;
            data_valid <= '1';
            state <= ST_HDR_C;
          elsif hdr_c_en = '1' then
            data_out   <= HEADER_C;
            data_valid <= '1';
            counter <= 0;
            state <= ST_SEND;
          else
            data_out <= data(0);
            data_valid <= '1';
            counter <= 1;
            state <= ST_SEND;
          end if;
        when ST_HDR_C =>
          if hdr_c_en = '1' then
            data_out   <= HEADER_C;
            data_valid <= '1';
            counter <= 0;
            state <= ST_SEND;
          else
            data_out <= data(0);
            data_valid <= '1';
            counter <= 1;
            state <= ST_SEND;
          end if;
        when ST_SEND =>
          data_out   <= data(counter);
          data_valid <= '1';
          if counter = NINPUTS-1 then
            state <= ST_IDLE;
          else
            counter <= counter + 1;
          end if;
        when others =>
          state <= ST_IDLE;
      end case;
    end if;
  end process st1;
end beh1;
```

Figure 8. Improved, but complex and hard to maintain one-process implementation

As we can see, it is difficult to efficiently implement the presented serializer using the standard methods of description of state machines. The desired solution should combine the clearness of the code offered by the standard implementations (Fig. 2 and Fig. 7) with the clock efficiency of the improved one-process implementation (Fig. 5).

It is necessary to mention, that the described problem may also be solved in a completely different way, e.g. by introducing of additional hardware which will detect if any header word is still to be sent, and returning the identifier of the first pending header word. In this case the state machine could use only three states: ST_IDLE, ST_SEND_HEADERS, ST_SEND_DATA. However this method leads to significant increase of complexity of the state machine and will be not discussed here.

4. PROPOSED SOLUTION

In the previously described solutions a separate state was allocated for checking of each possible header word. So the clock pulse is needed to move to the next state. The problem could be solved easily, if the machine could analyze the conditions for a few states in the same clock cycle. If the conditions for the particular state are met, the machine takes the defined actions and waits for the next clock pulse. If the conditions are not met, the machine continues checking of conditions for the next state (this will be called the "fall through" functionality).

The above functionality may not be implemented using the standard, signal based methods of description of state machines. The signal assignment is performed only at the end of the process, so there is no way to implement the "fall through" functionality. However this goal may be achieved by using of variables. The use of variables in synthesizable code is generally discouraged, but some developers report successfully using of them.[2]

The solution proposed in this paper is called "variable driven data flow in the sequential process". In this approach the variable is used to store the calculated next state of the machine, and only at the very end of the process the variable's value is assigned to the signal determining the next state of the machine. Because variable assignment is performed immediately, it is possible to use the new value in the rest of the process. However some additional changes are also needed. The big "case" instruction used in a standard implementation of state machine must be replaced with the sequence of "if" instructions, allowing us to implement the requested "fall through" feature. The last change is needed to provide a method for disabling of "fall through" feature if any activity requiring waiting for a next clock pulse is triggered. A dedicated "v_nowait" variable is used to control the "fall through" feature - it works only as long, as this variable is equal to "true". After the conditions associated with the particular state are met, and any operation requiring waiting till the next clock pulse is triggered, this variable should be set to false. Failing to do so may lead to incorrect and difficult to debug code.

The general structure of the code using the "variable driven flow control in sequential process" is very similar to the standard two-process implementation (please compare Fig. 7 and Fig. 9) and does not require duplicating of code. The serializer implemented with this method does not lose clock cycles for any combination of header enable bits (Fig. 6).

5. IS THE PROPOSED CODE SYNTHESIZABLE?

When using any more advanced techniques in writing of VHDL code, it is easy to produce a non-synthesizable code. It is possible, that the code which is a correct VHDL and which simulates perfectly, may not be converted into working hardware. Some of these limitations depend on the synthesis tools used, but some of them (e.g. when using constructs like "wait for 10 ns", or using two different clocks in the same sequential process) may not be synthesized by any tool. So the important question is if the presented code is synthesizable. To answer this question, the presented examples have been synthesized using the Altera's Quartus® II Web Edition ver. 6.0 SP1 tool. For the version based on "complex one-process implementation" (Fig. 8) the synthesized design has occupied 105 logic elements, and the maximum clock frequency was equal to 276.70 MHz. For the version based on "variable driven flow control in sequential process" (Fig. 9) the synthesized design has occupied 99 logic elements, and the maximum clock frequency was equal 316.36 MHz. The results of the post-synthesis simulation of the proposed design are shown in the Fig. 10. The obtained results prove, that the proposed method allows to produce synthesizable code with good properties.

However a care should be taken and post-synthesis simulation should be performed, especially if someone uses any older synthesis tools. A few years ago, when I first tried to use the described method, it was common, that some synthesis tools have generated incorrect implementation.

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

entity test1 is
  port (
    hdr_a_en   : in  std_logic;
    hdr_b_en   : in  std_logic;
    hdr_c_en   : in  std_logic;
    data       : in  T_INPUT;
    data_out   : out std_logic_vector(7 downto 0);
    data_valid : out std_logic;
    run        : in  std_logic;
    clk        : in  std_logic;
    rst        : in  std_logic);
end test1;

architecture beh1 of test1 is

  type T_STATE is (ST_IDLE, ST_HDR_A, ST_HDR_B,
                   ST_HDR_C, ST_SEND);
  signal state, state_next : T_STATE := ST_IDLE;
  signal counter,
    counter_next : integer range 0 to NINPUTS-1 := 0;
  signal run_s : std_logic;

begin  -- beh1

  -- sequential process
  st1s : process (clk, rst)
  begin  -- process st1s
    if rst = '0' then
      -- asynchronous reset (active low)
      state   <= ST_IDLE;
      counter <= 0;
      run_s   <= '0';
    elsif clk'event and clk = '1' then
      -- rising clock edge
      state   <= state_next;
      counter <= counter_next;
      run_s   <= run;
    end if;
  end process st1s;

  -- combinatorial process
  st1c : process (counter, data, hdr_a_en, hdr_b_en,
                  hdr_c_en, rst, run_s, state)
    variable v_next  : T_STATE;
    variable v_nwait : boolean := true;
  begin  -- process st1c
    -- defaults
    data_valid   <= '0';
    data_out     <= x"00";
    v_next       := state;
    v_nwait      := true;
    counter_next <= counter;
    if rst = '1' then
      -- normal operation
      if v_nwait and v_next = ST_IDLE then
        counter_next <= 0;
        if run_s = '1' then
          v_next := ST_HDR_A;
        end if;
      end if;
      if v_nwait and v_next = ST_HDR_A then
        if hdr_a_en = '1' then
          data_out   <= HEADER_A;
          data_valid <= '1';
          v_nwait    := false;
        end if;
        v_next := ST_HDR_B;
      end if;
      if v_nwait and v_next = ST_HDR_B then
        if hdr_b_en = '1' then
          data_out   <= HEADER_B;
          data_valid <= '1';
          v_nwait    := false;
        end if;
        v_next := ST_HDR_C;
      end if;
      if v_nwait and v_next = ST_HDR_C then
        if hdr_c_en = '1' then
          data_out   <= HEADER_C;
          data_valid <= '1';
          v_nwait    := false;
        end if;
        v_next := ST_SEND;
      end if;
      if v_nwait and v_next = ST_SEND then
        data_out   <= data(counter);
        data_valid <= '1';
        if counter = NINPUTS-1 then
          v_next := ST_IDLE;
        else
          counter_next <= counter + 1;
        end if;
      end if;
    end if;
    state_next <= v_next;
  end process st1c;

end beh1;
```

Figure 9. Two process implementation using the “variable driven flow control in sequential process” technique

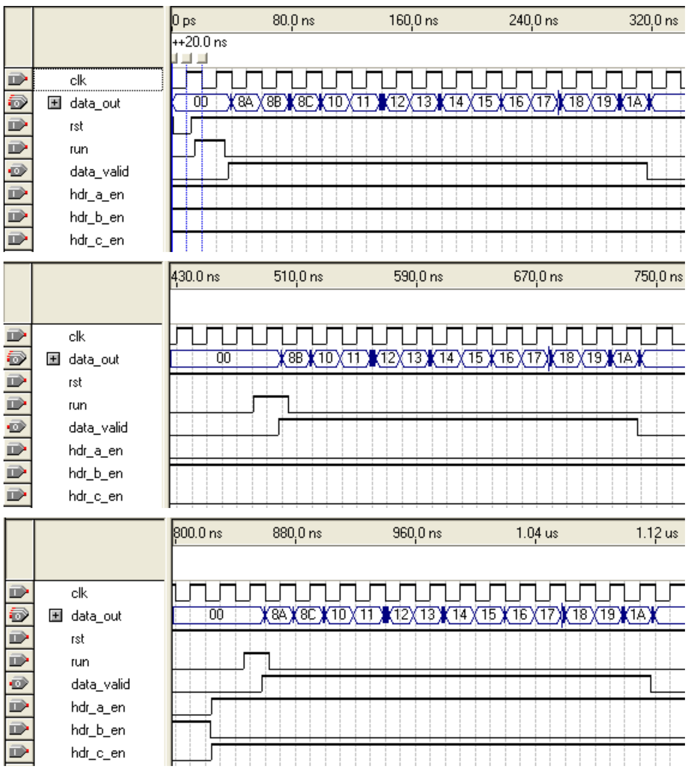

Figure 10. The functional simulation waveforms in the implementation with the "variable driven flow control in sequential process".

6. CONCLUSIONS

A new method for description of some state machines in the VHDL code has been proposed. The presented method offers serious advantages when a standard approach does not allow to obtain both good code readability and good performance of resulting implementation. The described method "variable driven flow control in sequential process" uses variables to efficiently describe complex state transitions without the need to duplicate code or to lose clock cycles when checking for multiple conditions in a sequence of states.

The presented method however requires a high coding discipline, as the introduced "fall through" functionality is potentially dangerous, and may lead to incorrect and difficult to debug code, when used improperly.

ACKNOWLEDGMENTS

The functional simulations presented in the paper have been performed using GHDL[3] - the free VHDL simulator. The diagrams have been prepared using the gtkwave[4] program. The synthesis and post-synthesis simulations have been performed using the Quartus® II Web Edition ver. 6.0 SP1 freely available from Altera.

REFERENCES

1. K. Skahill, *VHDL for Programmable Logic*, Prentice Hall, 1996.
2. J. Gaisler, "A structured VHDL design method." <http://www.gaisler.com/doc/vhdl2proc.pdf>.
3. "GHDL, free VHDL simulator." <http://ghdl.free.fr>.
4. "gtkwave - visualization tool for vcd, lxt, and vzt files." <http://home.nc.rr.com/gtkwave/>.