Introduction to VHDL
Case Study

Ilya Dmitrichenko

January 12, 2011

Preface

The task of this homework was to learn VHDL design and verification by example. A selection of standard digital circuits has been provided, of which a number of combinational and multifunction entities were implemented.

Code verification and simulation were performed using *Altera Quartus II* software suite with *Mentor Graphics ModelSIM (AE 6.5)* simulator. Only basic RTL simulation had been carried out, without any timing constraints applied. The *ModelSIM* test waveforms were generated using either GUI or Tcl scripts, VHDL testbenches were designed for some of the entities.

The fallowing two chapters are describing the implementation for each of the design units, fallowed by source code listing[1] and simulation results.

[1] The comments were removed, since all the information is provided in the text. The source code can be also viewed and downloaded from online repository at the fallowing URL: *https://github.com/errordeveloper/vhdl-misc-ct3032n/*

Contents

List of Figures

Chapter 1

Design Units: Combinational Logic

1.1 Comparator

1.1.1 Description

Comparator is a generic logic circuit that has two input ports and three single-bit outputs. Only one of each of the outputs may be asserted high at any given time. This design entity has two 4-bit inputs (A & B). When $A < B$, output L is high; when $A = B$ then E is high; $A > B$ then G is high.

Behavioural VHDL implementation of this uses `if-elsif` conditions. Currently it will only work for unsigned numbers of arbitrary bit-width.

1.1.2 Verification

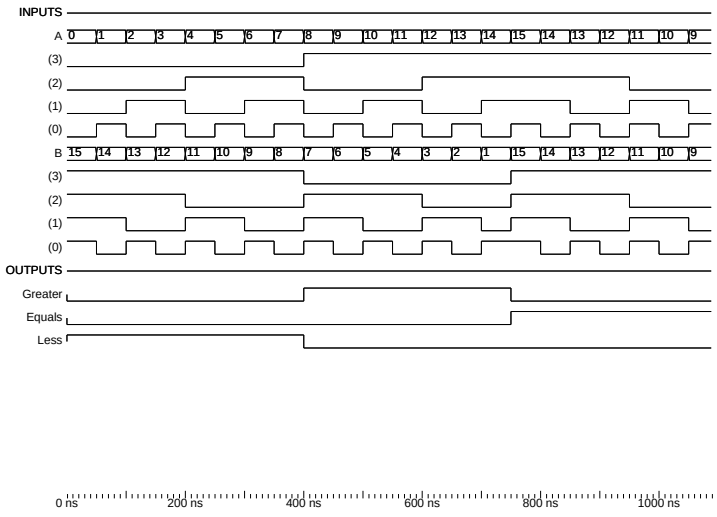


Figure 1.1: Simulation output with counter sequences applied to 4-bit comparator

```
Library IEEE;
    use IEEE.STD_LOGIC_1164.all;

Entity COMP4 is
  PORT (
        A: in  std_Logic_Vector ( 3 DOWNTO 0 );
        B: in  std_Logic_Vector ( 3 DOWNTO 0 );
        G: out std_Logic;
        L: out std_Logic;
        E: out std_Logic
       );
end entity;

Architecture Behaviour of COMP4 is
begin

  compare: PROCESS ( A, B ) begin

        If ( A = B ) then
           G <= '0';
           L <= '0';
           E <= '1';
        elsif ( A < B ) then
           G <= '0';
           L <= '1';
           E <= '0';
        elsif ( A > B ) then
           G <= '1';
           L <= '0';
           E <= '0';
        end if;

  end process;

end architecture;
```

Listing 1.1: *Behavioural implementation of 4-bit Comparator*

1.2 Priority Encoder

1.2.1 Description

Priority Encoder (PE) is a generic combinational circuit that is most commonly used to attach several inputs to one external interrupt pin of a microcontroller. Taking a number of lines, it outputs only one input value based on priority. If there is an event at input $V(0)$, then other lines are ignored. If line $V(0)$ is quite, then input $V(1)$ is checked, the last line $V(n)$ will be output only when $V(n-1)$ and all other lines are quite.

PE also has an output for the address of currently active line. This design entity has four input lines, and uses 2-bit addressing. Note that the address of previously active line will be held until it changes, otherwise an extra bit would be needed to implement addressing of inactivity. A procedure *`maskf()`* had been coded to facilitate addressing using casting from integer argument to logic vector. This procedure takes vector V, masks one bit M and outputs $V(M)$ to the bit vector bus X, writing the binary value of masking bit M to the address vector bus W. The event condition checking has to be done prior the procedure call.

1.2.2 Verification

The code in listing 1.2 demonstrates two valid design options and one invalid.

This VHDL design uses `if-generate` conditions depending on value of `generic` variables, alternatively multiple architectures could be coded, but that would take more lines of code. The invalid version uses *"don't care"* `case` branching, though this approach cannot be synthesised. Two very similar alternatives are shown in the listing (1.2). It is probably most straight-forward way to use *`std_match()`* function from `IEEE.NUMERIC_STD` library.

Only level-triggering is implemented, since use of several *`rising_edge()`* conditions introduces multiple clocking issues, which are hard to overcome.

The simulation output is shown in figure 1.2.

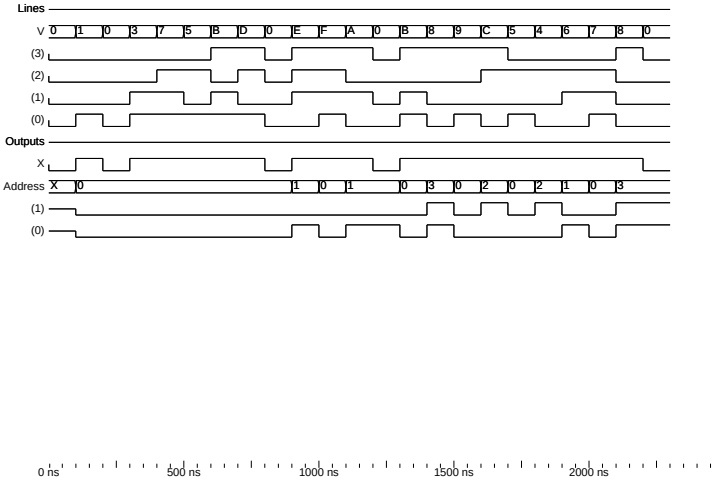


Figure 1.2: Random stimulus applied to inputs lines of 4-bit PE

```
Library IEEE;
    use IEEE.STD_LOGIC_1164.all;
    use IEEE.STD_LOGIC_UNSIGNED.all;
    use IEEE.NUMERIC_STD.all;

Entity PE4 is

  GENERIC (dopt_logic_1: boolean := false;
           dopt_logic_2: boolean := true;
           dopt_invalid: boolean := false);

  PORT (V:      in      std_Logic_Vector ( 3 DOWNTO 0 );
        A:      out     std_Logic_Vector ( 1 DOWNTO 0 );
        X:      out     std_Logic);

 procedure maskf (
    signal V:   in std_Logic_Vector(3 DOWNTO 0);
    constant M: natural;
    signal X:   out std_Logic;
    signal W:   out std_Logic_Vector(1 DOWNTO 0)) is
 begin
        X <= V(M);
        W <= STD_LOGIC_VECTOR(TO_UNSIGNED(M, 2));
 end procedure;

end entity;

Architecture Behavioural of PE4 is

  SIGNAL W: std_Logic_Vector ( 1 DOWNTO 0 );
  SIGNAL Q: std_Logic;

 begin

logic_1: if dopt_logic_1 generate
PROCESS ( V ) begin

        If ( V(0) = '1' ) then
                maskf(V, 0, Q, W);

        Elsif ( V(1) = '1' ) then
                maskf(V, 1, Q, W);

        Elsif ( V(2) = '1' ) then
                maskf(V, 2, Q, W);

        Elsif ( V(3) = '1' ) then
                maskf(V, 3, Q, W);
```

```
        Else Q <= '0';
        end if;

end process; end generate;

logic_2: if dopt_logic_2 generate
PROCESS ( V ) begin

    If ( std_match(V, "---1") ) then
        maskf (V, 0, Q, W);

    Elsif ( std_match(V, "--10") ) then
        maskf (V, 1, Q, W);

    Elsif ( std_match(V, "-100") ) then
        maskf (V, 2, Q, W);

    Elsif ( std_match(V, "1000") ) then
        maskf (V, 3, Q, W);

        else Q <= '0';

    end if;

end process; end generate;

invalid: if dopt_invalid generate
PROCESS ( V ) begin

  case V is
        when "---1" => Q <= V(0); W <= "00";
        when "--10" => Q <= V(1); W <= "01";
        when "-100" => Q <= V(2); W <= "10";
        when "1000" => Q <= V(3); W <= "11";
        when others => Q <= '0';  W <= "XX";
  end case;

end process; end generate;

   A <= W;
   X <= Q;

end architecture;
```

Listing 1.2: *Behavioural implementation of 4-bit PE*

1.3 Arithmetic Logic Unit

1.3.1 Description

Arithmetic Logic Units (ALUs) are some of the most common building block in any processor architecture.

This design entity has three 4-bit inputs (A & B), operation instruction input (I) and result output (X). This is a combinational logic circuit and does not require a clock.

Please note, the carry bits are ignored[1]. *Multiplication* and *division* are not yet implemented[2], but instructions `'0x2'` and `'0x3'` had been reserved. All logic operations (except *inversion*[3]) were implemented.

Below is the list of instructions for this ALU.

- `Arithmetic:`

  `'0x0'` *addition*

  `'0x1'` *subtraction*

- `Logic:`

  `'0x4'` *OR*

  `'0x5'` *XOR*

  `'0x6'` *XNOR*

  `'0x7'` *NOR*

  `'0x8'` *AND*

  `'0x9'` *NAND*

In addition to `IEEE.STD_LOGIC_1164`, this entity requires fallowing libraries:

- `IEEE.STD_LOGIC_ARITH` for arithmetic operations
- `IEEE.STD_LOGIC_SIGNED` for operations with unsigned integers

1.3.2 Verification

The code in lisling 1.3 had been synthesised using and the Register Transfer Level (RTL) circuit is shown in figure 1.5. To verify correct operation of this design, a stimulus needs to be generated. Applying a count sequence may appear to be appropriate, though it would produce numerous waveform figures. Though a random wave may be used, it would make results harder to read on paper.

A VHDL testbench had been generated and edited for the ALU simulation[4]. See figures 1.3 and 1.4 for the output waveforms.

[1] Carry bit can be implemented by adding an extra 1-bit vector input `(0 DOWNTO 0)` and by summing two 4-bit vectors with 3rd 1-bit vector the VHDL synthesis tool is expected to recognise it as the carry input. For simplicity of the design, this had been omitted.

[2] *Multiplication* needs 8-bit vector to output the product and *division* implies conversion to floating point or integer approximation.

[3] *Inversion* operates on one input only.

[4] The testbench code is too long to be included in the report, it can be viewed at the fallowing URL: *https://github.com/errordeveloper/vhdl-misc-ct3032n/blob/master/code/comblogic/alu_4bit.vht*

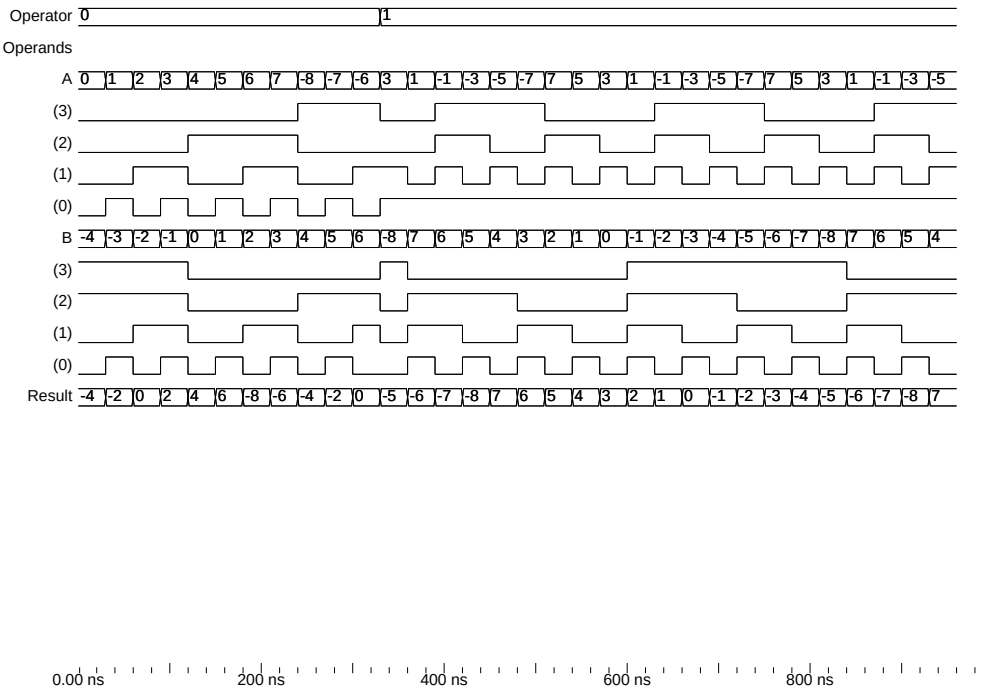


Figure 1.3: Test result of arithmetic modes

Figure 1.3 shows `for-loop` test in *addition* and *subtraction* modes. The overflow occurs at rather low values, proving that this ALU is not operational for arithmetics with result values greater then +6 or less then −8.

Waveforms in figure 1.4 show the logic mode test results applying same input bit sequences. The testbench used a `for-loop` to cycle through the operator commands. The results were manually checked and found to be correct.

Conclusion

A 4-bit ALU without a carry output is rather unusable for arithmetics. This design would be useful as a building block to produce 8-, 12-, 16-, 24-bit and greater width units. However, there is no need for such component in VHDL, since only a minor modification to the ALU code would be required to increase the capabilities. Thought, it is not as simple when hardware constraints are in place, because many different adder design pattern are available and the decision has to be made in terms of physical area, performance or power optimisation preferences made before loading the circuit design into programmable devices.

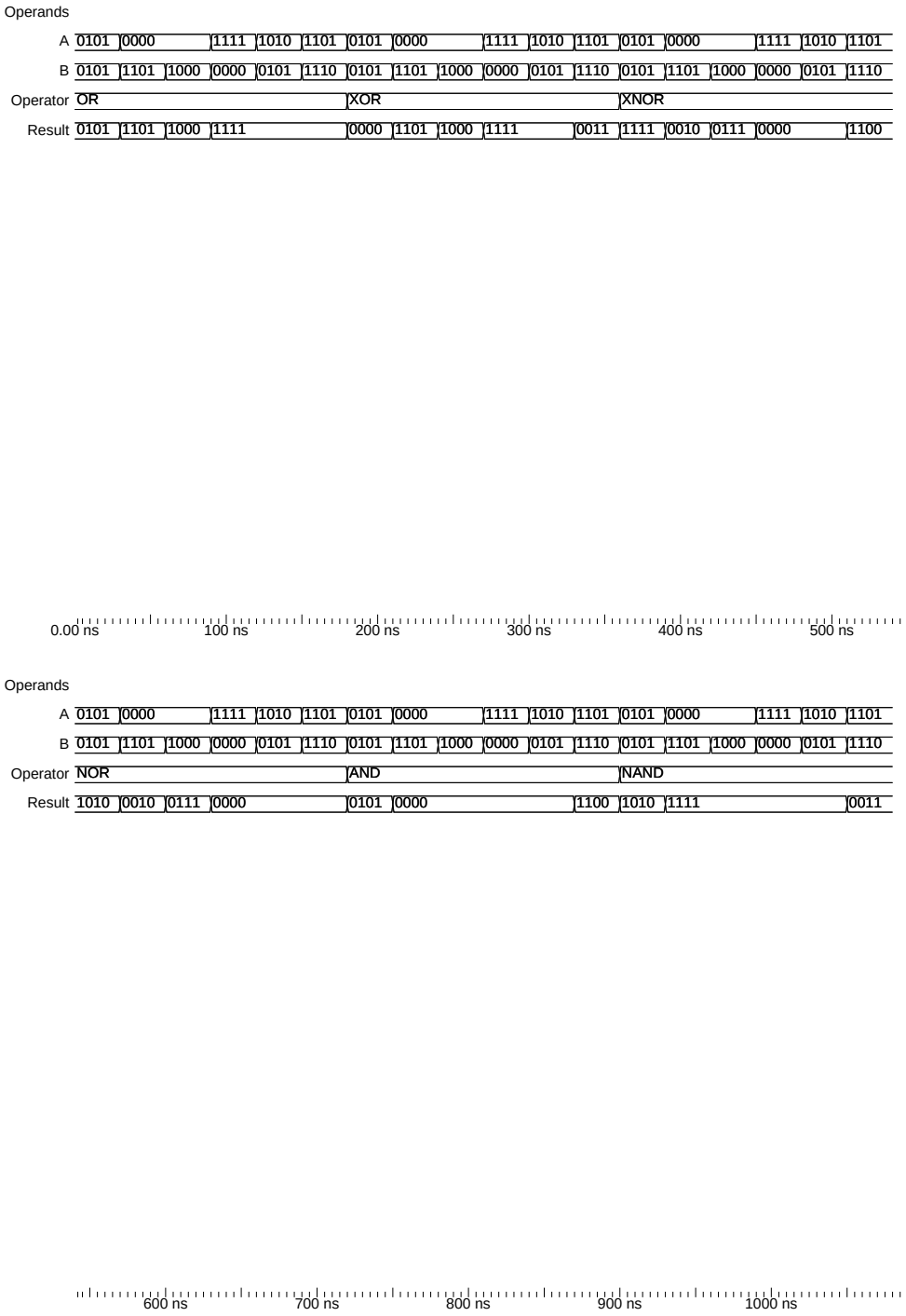


Figure 1.4: Test result of logic operations

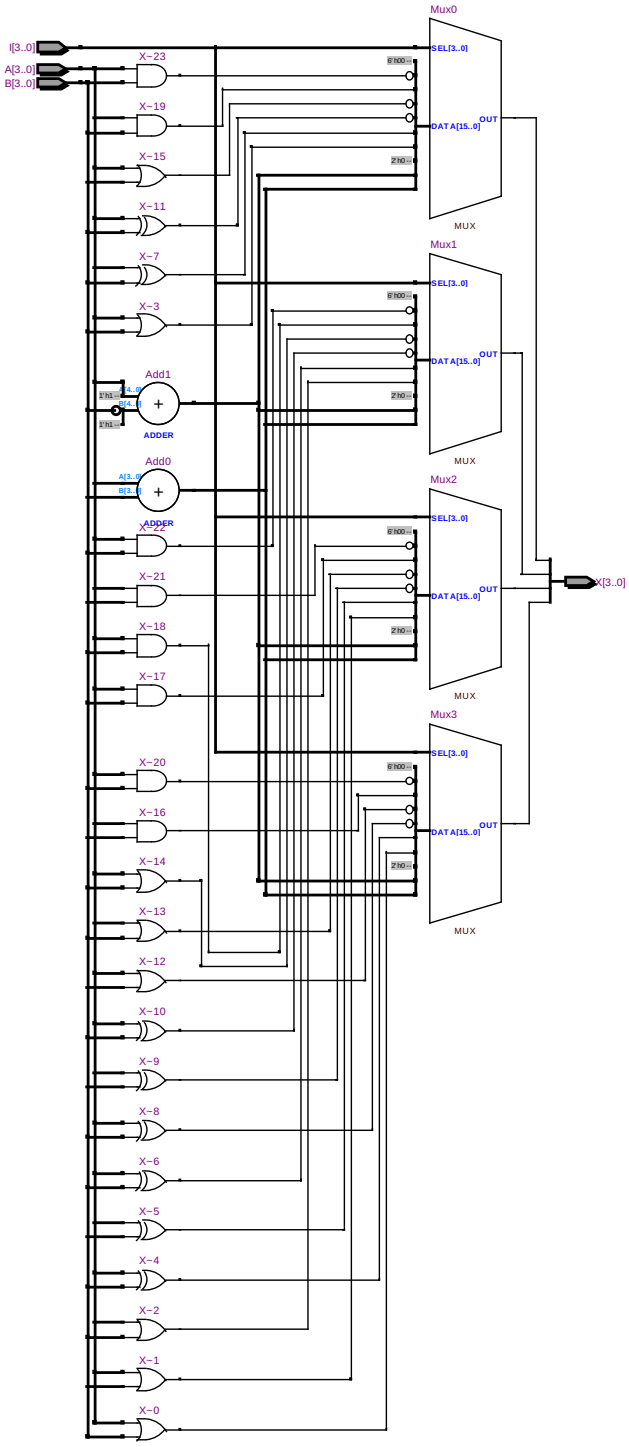


Figure 1.5: RTL circuit diagram for ALU

```
Library IEEE;
    use IEEE.STD_LOGIC_1164.all;
    use IEEE.STD_LOGIC_ARITH.all;
    use IEEE.STD_LOGIC_SIGNED.all;

Entity ALU is
  PORT (
        A: in  std_Logic_Vector ( 3 DOWNTO 0 );
        B: in  std_Logic_Vector ( 3 DOWNTO 0 );
        I: in  std_Logic_Vector ( 3 DOWNTO 0 );
        X: out std_Logic_Vector ( 3 DOWNTO 0 )
       );
end entity;

Architecture Behavioral of ALU is
begin

  PROCESS ( A, B, I)

  begin
        case I is

          when "0000" => X <= (A + B);

          when "0001" => X <= (A - B);

          when "0100" => X <= ( A OR B );

          when "0101" => X <= ( A XOR B );

          when "0110" => X <= ( A XNOR B );

          when "0111" => X <= ( A NOR B );

          when "1000" => X <= ( A AND B );

          when "1001" => X <= ( A NAND B );

          when others => X <= "XXXX";

        end case;

  end process;

end architecture;
```

Listing 1.3: *Behavioural implementation of 4-bit ALU*

Chapter 2

Design Units: Multifunctional Logic

2.1 4-bit Register

2.1.1 Description

Figure 2.1 shows a block diagram with inputs and outputs of the register. The opcodes for `Control` inputs are as fallows:

`Enable`	`Control`	*Command*	*Function*
`0`	`XX`	*HOLD*	$Q^{n+1} = Q^n$
`1`	`00`	*CLEAR*	$Q^{n+1} = 0$
`1`	`11`	*LOAD*	$Q^{n+1} = D^n$
`1`	`01`	*OR BITS*	$Q^{n+1} = Q^n \cdot D^n$
`1`	`10`	*AND BITS*	$Q^{n+1} = Q^n \oplus D^n$

2.1.2 Verification

To simulate the register behaviour a simple testbench had been coded. The waveform in figure 2.3 demonstrates that changes in the output occur synchronously and the operation of this model does correspond to the given specification. For example, the output doesn't clear until `Enable` input goes ***high*** disregarding the state of `Control` input, also the output transaction doesn't occur until the rising edge of the `Clock` signal. Logic operations are also correct and synchronous. The RTL diagram (figure 2.2) appears to be very linear, nevertheless during HDL design of this model there had been a few errors made when the RTL viewer came in handy.

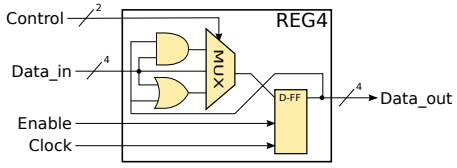


Figure 2.1: Multifunctional register block diagram

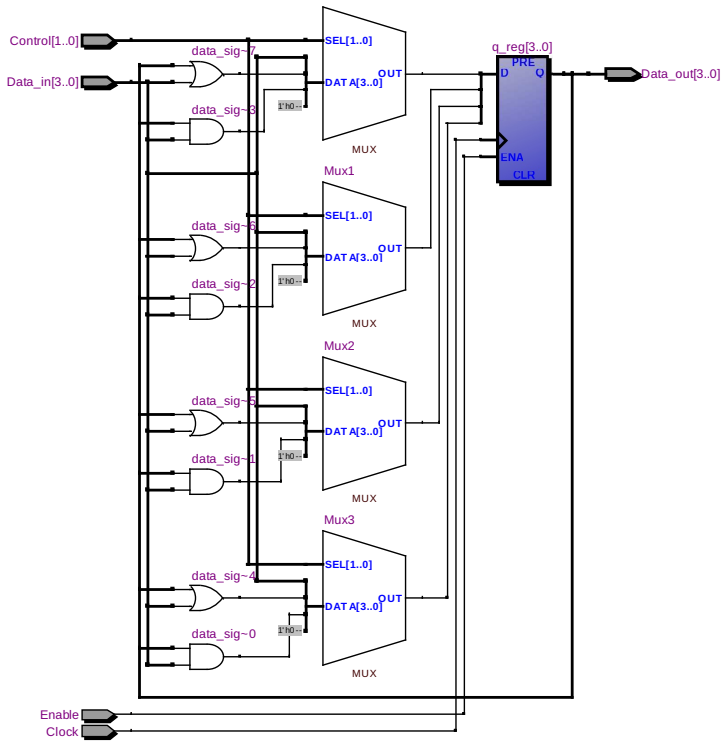


Figure 2.2: Multifunctional register RTL circuit diagram

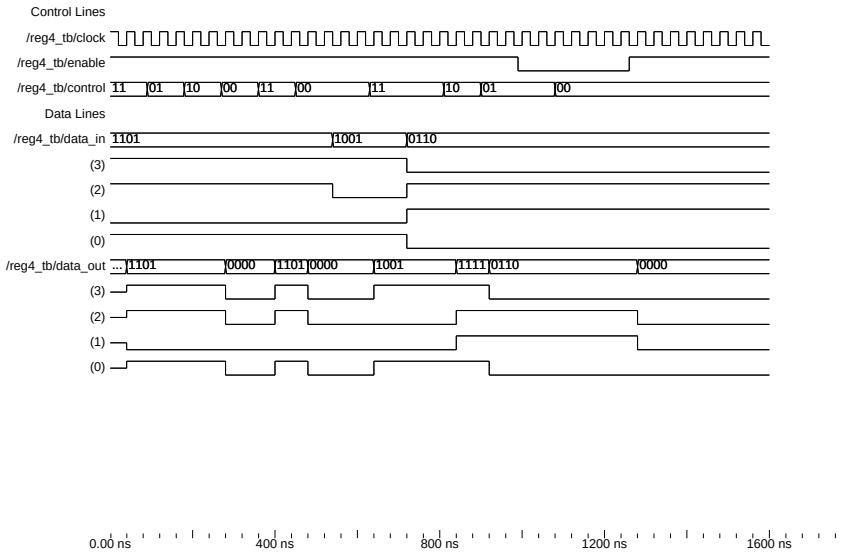


Figure 2.3: Multifunctional register simulation

```
Entity REG4 is
-- PORTS : Clock, Enable, Control, Data_in & Data_out
end entity REG4;

Architecture Behavioural of REG4 is
 signal q_reg, q_next,
        data_sig: std_Logic_Vector ( 3 DOWNTO 0 );
begin

PROCESS ( Clock ) begin

 If ( rising_edge(Clock) ) then
        -- Clasic sync construct
        q_reg <= q_next; end if;

end process;
  -- Output the data when enabled
  q_next <= data_sig when Enable = '1' else
          -- Hold current output when disabled
          q_reg;

PROCESS ( Control ) begin

  case Control is

     when "00" => -- Clear the output
           data_sig <= ( others => '0' );

     when "01" => -- AND operation
           data_sig <= ( q_reg AND Data_in );

     when "10" => -- OR operation
           data_sig <= ( q_reg OR Data_in );

     when "11" => -- Load the data
           data_sig <= ( Data_in );

     when others => -- Don't care
           data_sig <= q_reg;

  end case;

end process;

  Data_out <= q_reg;

end architecture;
```

Listing 2.1: *Behavioural implementation of 4-bit register (brief)*

2.2 8-bit Counter

2.2.1 Description

An attempt to implement multifunction counter had been made. It is featuring two 8-bit wide ports: count output and increment input, 2-bit mode control as well as *mode-sensitive* overflow indicator.

In regards to the width of the `Increment` port, there no particular need to have 8-bit, however the VHDL synthesis tools will apply an *8x8-adder* if the increment value of the counter had been shorter.

The `Terminate` pin is *mode-sensitive* overflow detector, it goes high one clock cycle before the `Count` output overflows. That is when it's `'0x00'` in *count-down (*`'0b01'`*)*, and `'0xff'` in *count-up (*`'0b10'`*)* mode.

- The `Control` port accepts fallowing opcodes:

  `'0b10'` *count-up*

  `'0b01'` *count-down*

  `'0b11'` *clear output to* 255

  `'0b00'` *clear output to* 0

Same convention for signal naming had been employed for in this code *(see listing below (2.2) for* `q_reg`*,* `q_next` *&* `c_sig`*).* Also the `Control` and `Increment` inputs are latched in order to prevent any sudden changes in these affecting the output stability. It may be a good idea to add `if-generate` conditions to disable or enable such behaviour. Also a few `generic` parameter need to be added to achieve fully re-usable library code.

2.2.2 Verification

The figures 2.4 and 2.5 are to approve correct operation of this 8-bit counter design. Various values of `Increment` input were forced in the simulation, and it would take several pages to print out those waveforms. Therefore below is the extract demonstrating the key performance factors. As mentioned above, the latching of the *Control* input is for stability purpose, though it can be seen as *"slowness"* – this is shown by the waveforms in figure 2.5.

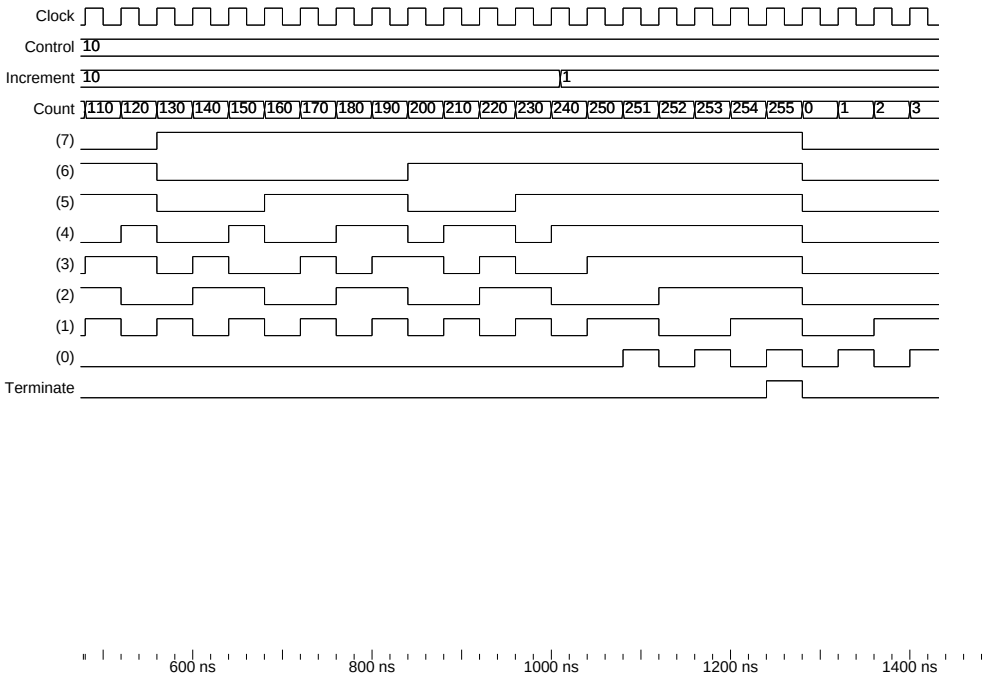


Figure 2.4: Counter simulation: up-count overflow region

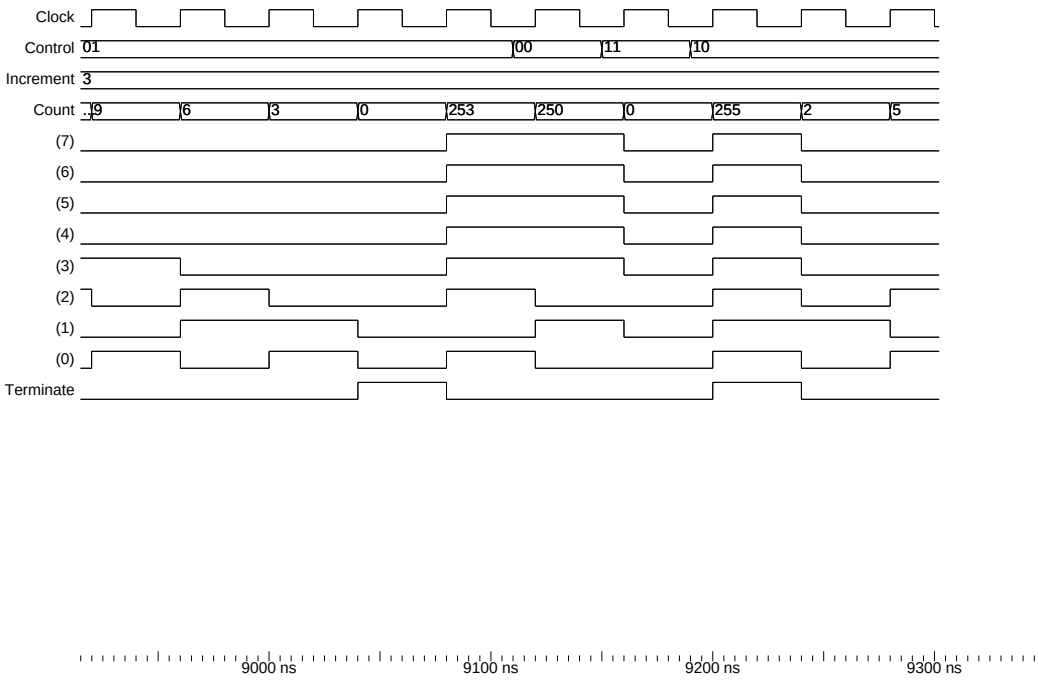


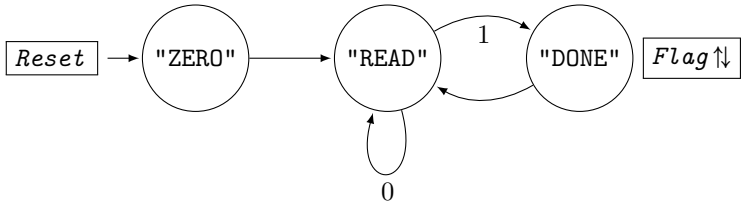
Figure 2.5: Counter simulation: down-count overflow region & 1-cycle latch delays

```
Entity C8 is
-- PORTS : Clock, Count, Control, Increment & Terminate
end entity C8;

Architecture Behavioural of C8 is

signal i_sig, q_reg, q_next:
       std_Logic_Vector ( 7 DOWNTO 0 );
signal c_sig: std_Logic_Vector ( 1 DOWNTO 0 );
signal t_sig: std_Logic;

begin

PROCESS ( Clock ) begin

    If ( rising_edge(Clock) ) then

        -- Clasic sync construct
        q_reg <= q_next;

        -- Latch the inputs
        c_sig <= Control;
        i_sig <= Increment;

    end if;

end process;

PROCESS ( c_sig, q_reg ) begin

  case c_sig is

     when "00" => -- Clear the output to '00'
          q_next <= ( others => '0' );

     when "01" => --  Count down
          q_next <= ( q_reg - i_sig );
          If q_reg = X"00" then
               t_sig <= '1';
          else t_sig <= '0';
          end if;

     when "10" => --  Count up
          q_next <= ( q_reg + i_sig );
          If q_reg = X"ff" then
               t_sig <= '1';
          else t_sig <= '0';
          end if;
```

```
      when "11" => -- Cler the output to 'ff'
            q_next <= ( others => '1' );

      when others => -- Don't care
            q_next <= q_reg;

  end case;


end process;

---- The fallowing signal assignment generates
---- inappropriate circuit, otherwise it would
---- ideal for the purpose of this signal.
-- Terminate <= '1' when ( q_next = (not q_reg) ) else
--               '0';

  Terminate <= t_sig;
  Count <= q_reg;

end architecture;
```

Listing 2.2: *Behavioural implementation of 8-bit counter (brief)*

Figure 2.6: FSM of Universal Sequence Detector



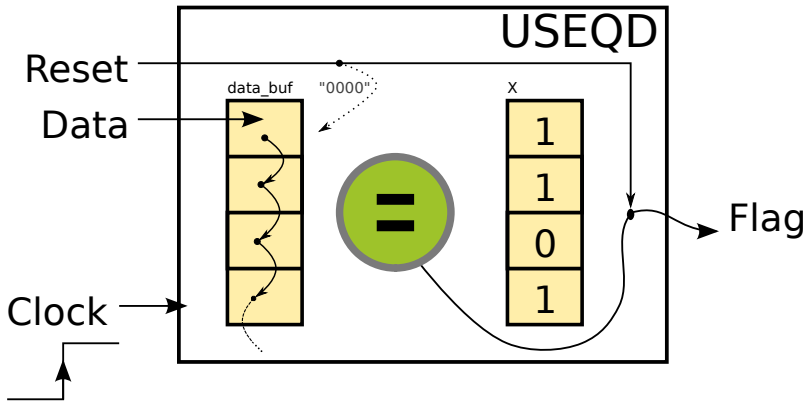


Figure 2.7: Block diagram

2.3 Universal Sequence Detector

2.3.1 Description

The sequence detector which is to be described and verified in this section is dedicated to have three inputs – serial `Data`, `Clock` and `Reset`, and one `Flag` output. The output is to rise *high* on the clock cycle after the given sequence had been received at the `Data` input. The `Reset` input (*active-high*) is to clear the state of the circuit and *Clock* triggers on the positive edge.

The details of implementation of this entity are discussed below, the two diagrams on this page (2.6 and 2.7) show the final design of the sequence detector, which has a Finite State Machine with three states, and it use a shift-register to store the serial data and match sequences with pre-defined bit vector.

Algorithm Considerations

Firstly it would be appropriate to contrast the VHDL design pattern with more classic programming. Below is a basic implementation for an equivalent algorithmic structure in a higher-level programming language. In terms of optimisation, if the pattern details are to be supplied at run-time, then there is no particular need for the code to be optimised. An attempt to do so may in fact increase the duration of execution. Suppose the values are know before the compilation, then a variety of techniques could be applied (including automated methods with scripting, pre-processor macros, compiler hacking or other).

```c
/** This function returns integer sample */
int getX( void );

#define L 7 /** Length of given sequence */

/** An interesting patter to be detected */
int Z[L] = { 123, 234, 345, 456, 567, 678, 790 };

int main ( void )
{
  int i;

  while(1) {

    for ( i = 0; i < L; i++ ) {

       if ( getX() == Z[i] ) continue;

       else break;
    }

    /** Check loop exit index */
    if ( i == L ) return 0;

    /** Will try forever */
    else continue;

  }

}
```

Listing 2.3: *Brief algorithm prototype of sequence detector in C*

The VHDL as language provide plenty of choices, for example some data types that it allows do not appear very low-level at first. And it then turns out that this language dictates certain restriction on how those types may be used. Same applies to various constructs (such as loops) and other elements of the language. Most of the code listing in this report are indeed examples of standard VHDL code patterns. Most of the above listings have not been intended as very generic (i.e. re-usable) peaces.

In this last section of the case study an attempt was made to produces the most generic design entity. This will result in code, which is not necessarily optimised to detect a particular sequence of bits. Thought, the VHDL synthesis tools are to accomplish this task, unless the system is to be configured during run-time (i.e. the sequence that is to be detected will be loaded as an input). The listings and simulation results below shall demonstrate how give a complexity N in VHDL gets optimized to a lesser complex result when synthesised.

2.3.2 Verification

A number of design decisions had been tested before the code listed in 2.4 had been developed. The figure 2.6 on the top of this section shows the transitions between three states.

One of tried approaches have used a counter variable to index the incoming serial bitstream on `Data` pin and compare it to internal vector X. This early version can be viewed in the on-line code repository[1] After this was proven unsuccessful, the idea was to try implementing a shift-register to provide buffering for the serial data, and therefore synchronise the two `process` block in this manner. It turned out that by using *'&' concatenation* operator inside of the condition branch with *'if(rising_edge(Clock))'*, provides an appropriate shift-register for de-serialising and buffering the data stream.

To minimise the dimension of the FSM segment, only 3-state machine is being used. There is no particular need for defining any additional state and in fact the `'zero'` state may be eliminated, then synthesised logic reduces to very simple circuit as show in figures 2.8 and 2.12. The number of bits in match vector X does not affect the overall shape of the FSM, it only increase the width of the data path. Although, it is still quite useful to include this `'zero'` state, since clearing of the shift-register and `Flag` bit may be needed in some applications. Using single-line *'std_match(data_buf,X)'* function call is perhaps the most appropriate method in the `'read'` state (it is also show in RTL diagrams 2.8 and 2.12 that this function reduces to very simple logic circuit block).

Simulation Results

The simulation had carried out for two version of this sequence detector.

Firstly, a 4-bit sequence `'0b0011'` had been applied. A pseudo-random stimulus vector had been fed into the simulator via *ModelSIM's Tcl* interactive console. The result waveforms from this session are in figures 2.10 and 2.11. The `Flag` output is asserted in `'done'` state when the `'0b0011'` sequence is detected in `'data_buf'`. Therefore operation of this entity is considered to be correct.

A 9-bit sequence `'0b010110011'` (`'0x0B3'`) had been applied in the second simulator session. Produced waveform would be too long to include here, therefore only two important regions had been captured for printing (see figures 2.14 and 2.15). These results also demonstrate fully-functional detection of now wider bit sequence that appear in the shift-register (`data_buf`) at 12000ns mark. It had been noticed that data endianess may need to be modified. The sequence which had been fed in simulation had to be in reverse order. But this is in general not an issue, since many digital systems handle data either way.

Conclusion

The simulation of code listed below had proven that arbitrary complexity is very straight-forward to implement in VHDL. It is probably possible to achieve different behaviour of when the output signal is being asserted, but this would involve an alternative approach, perhaps adding an extra clock with faster rate. That is outside of the purpose of this report and depends on the application.

[1] Earlier code revision: `https://github.com/errordeveloper/vhdl-misc-ct3032n/blob/89419c215dbf80138a66ae61bbff5b5aca3976e0/code/multifunc/seq_fsm.vhd`

```
Library IEEE;
    use IEEE.STD_LOGIC_1164.all;
    use IEEE.NUMERIC_STD.all;

Entity USEQD is

GENERIC (
constant L: natural := 9 ;
constant X: std_Logic_Vector(L-1 DOWNTO 0) := "010110011"
    );

PORT ( Data: in std_Logic;
       Flag: out std_Logic;
       Reset: in std_Logic;
       Clock: in std_Logic );

end entity; -- SEQ;

Architecture FSM_BD of USEQD is

  type state_type is ( zero, read, done );

  signal state_next, state_reg: state_type;

  signal data_buf: std_Logic_Vector(L-1 DOWNTO 0) := (
     others => 'X');

begin


-- Events control process
events: PROCESS ( Clock, Reset ) begin

  If ( Reset = '1' ) then

    state_reg <= zero;

    data_buf <= ( others => '0' );

  elsif ( rising_edge(Clock) ) then

    state_reg <= state_next;

    data_buf <= ( Data & data_buf(L-1 DOWNTO 1) );

  end if;

end process; -- events
```

```
-- State control process
states: PROCESS ( data_buf, state_reg ) begin

 case state_reg is

  when read =>

   Flag <= '0';

   if ( std_match( data_buf, X ) ) then

     state_next <= done;

   else

     state_next <= read;

   end if;

  when done =>

   Flag <= '1';

   state_next <= read;

  when zero =>

   Flag <= '0';

   state_next <= read;

  when others =>

   state_next <= read;

 end case;

end process; -- states

end architecture; -- FSM;
```

Listing 2.4: *FSMD implementation of Universal Sequence Detector*

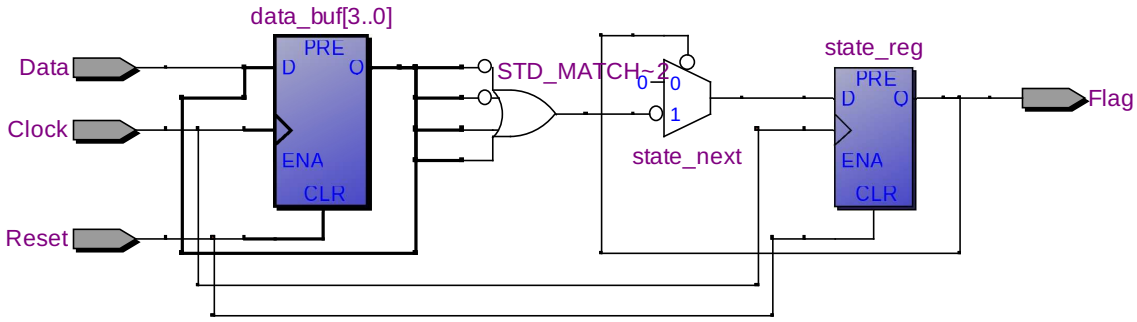


Figure 2.8: RTL view: Basic USEQD design (no ‘zero‘ state, for ’0*b*0011’)

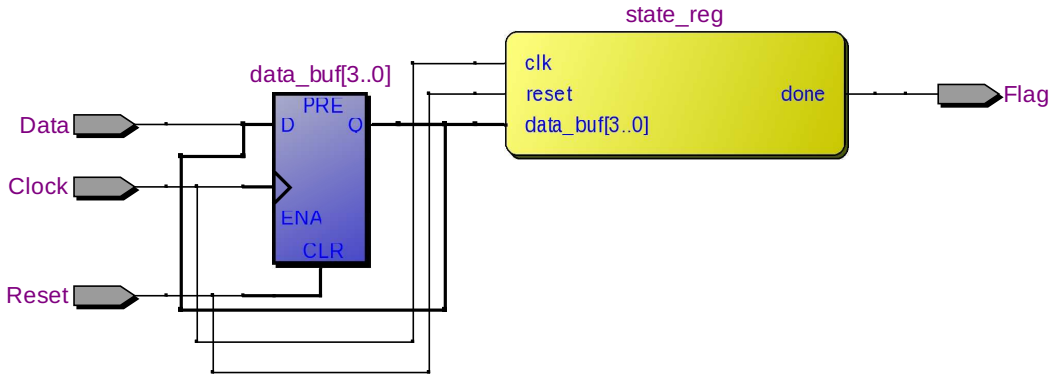


Figure 2.9: RTL view: Full USEQD using FSM with buffered data input (for ’0*b*0011’)

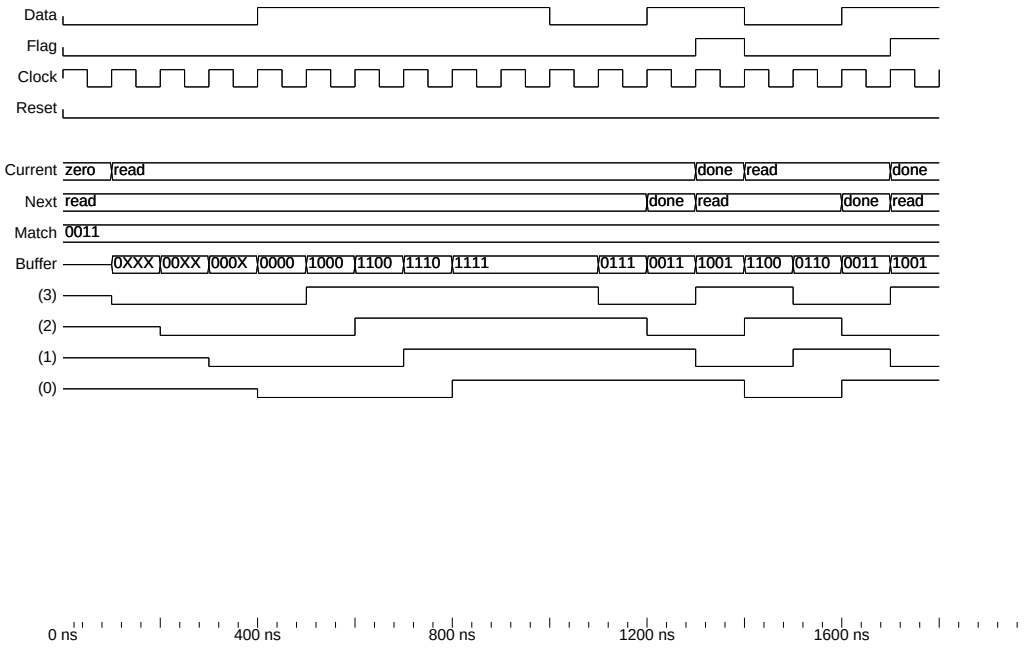


Figure 2.10: Detecting sequence '0*b*0011'

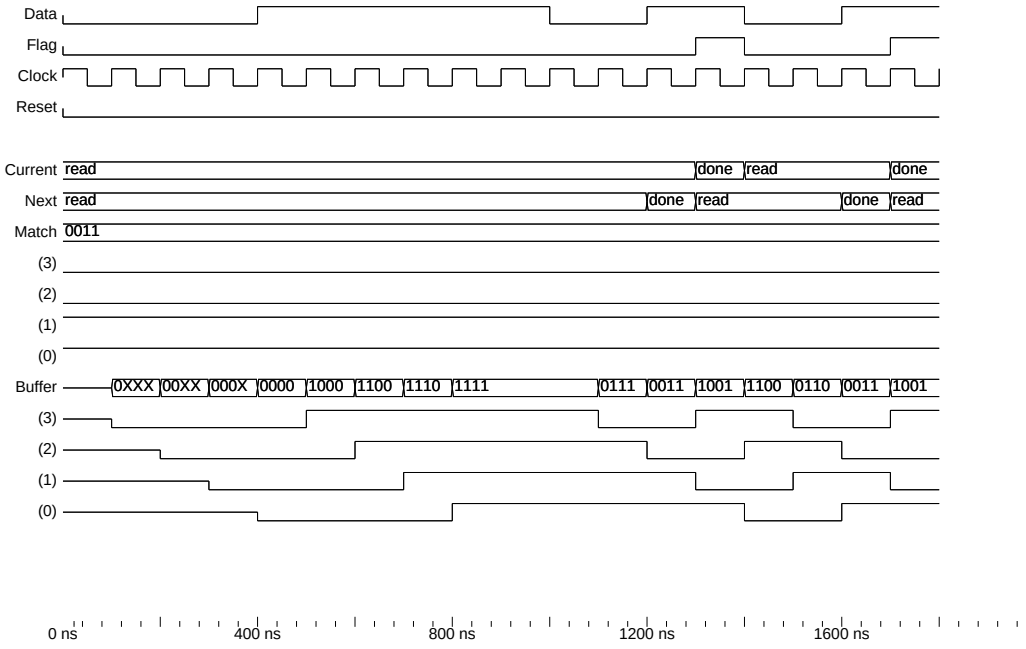


Figure 2.11: Detecting sequence '0*b*0011' (with basic circuit)

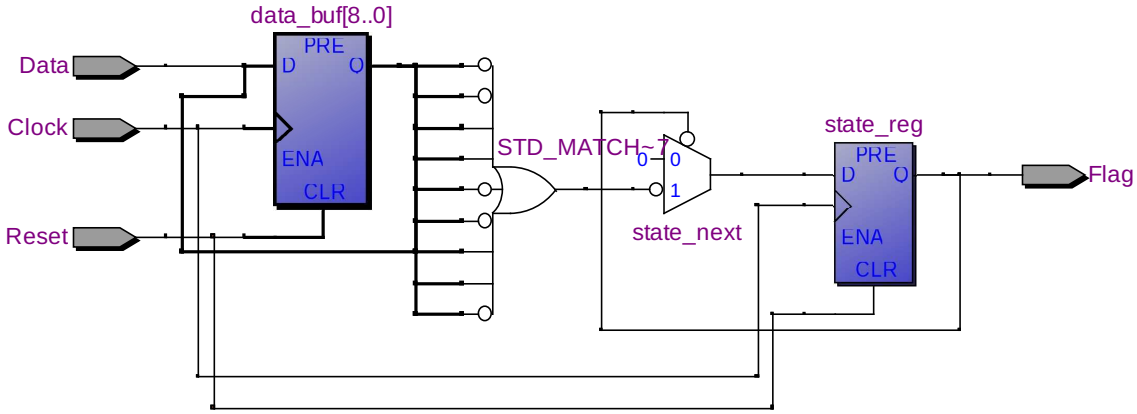


Figure 2.12: RTL view: Basic 9-bit USEQD design

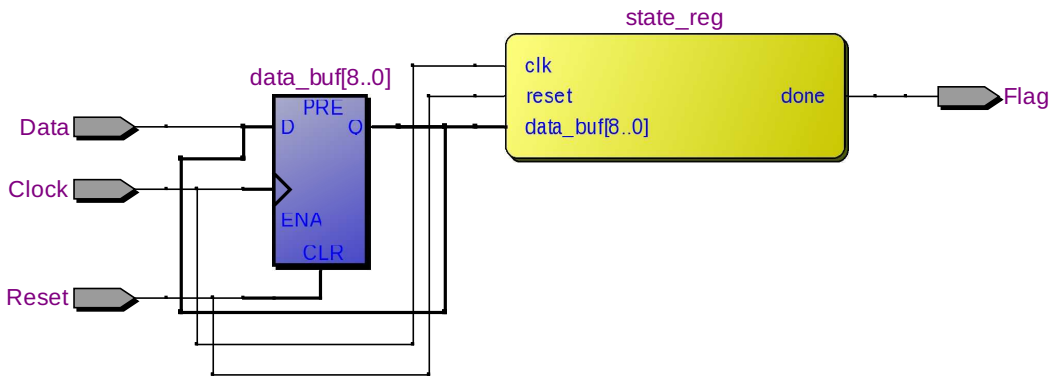


Figure 2.13: RTL view: Full 9-bit USEQD using FSM with buffered data input

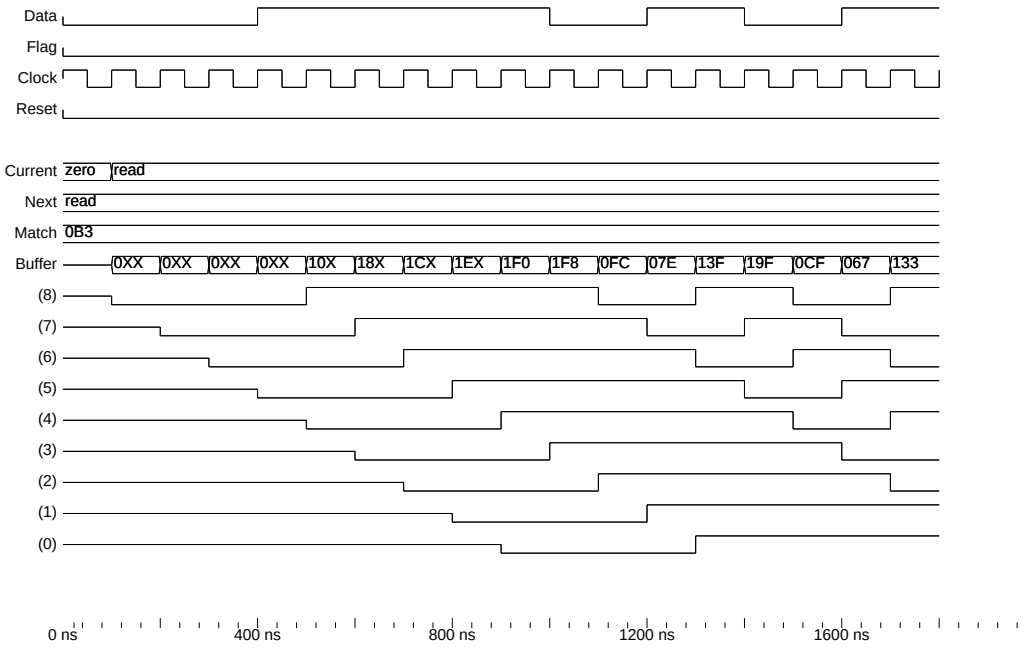


Figure 2.14: Detecting sequence '0*b*010110011' (without match)

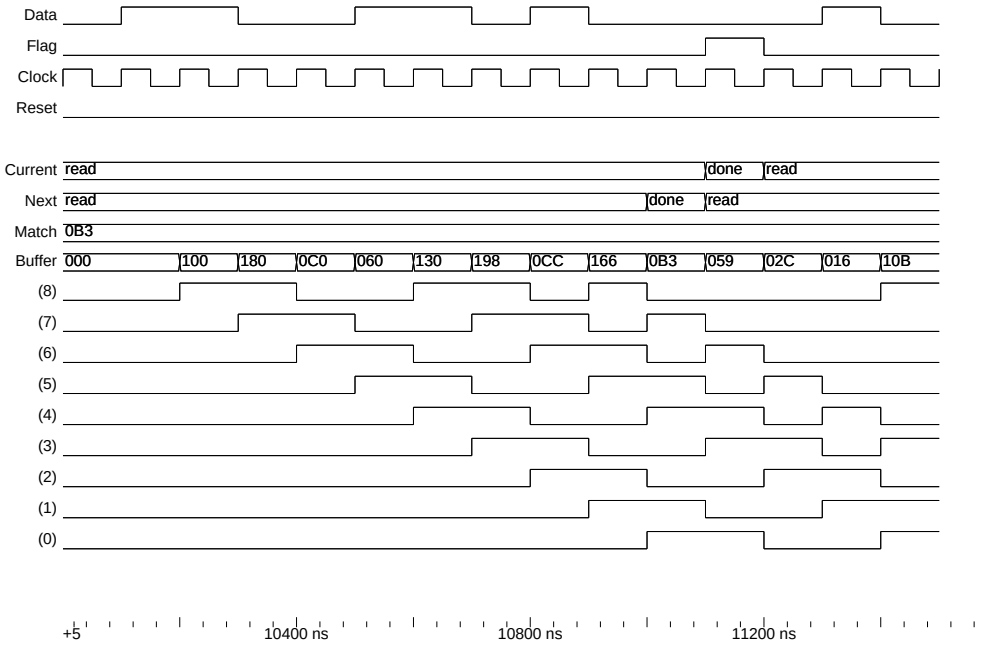


Figure 2.15: Detecting sequence '0*b*10110011'

Chapter 3

Conclusion

Learning Outcome

During completion of this case study a number of techniques were exercised in VHDL design, verification and simulation. As it had been already mentioned the software used to aid the workflow is *Altera Quartus II* and *ModelSIM (Altera Edition)*. Few other packages which were tested during the case study include *GTKwave* and *Doxygen*, but the output was not used in this report.

The aspects of behavioural and FSM design were studied, utilising IEEE standard logic and numeric libraries. Some of the major subjects which are still to be studied are timing analysis and physical aspects of FPGAs. In the next stage few of the above designs should be packaged for later re-use.

Links

- Case Study Source Code Repository
  `https://github.com/errordeveloper/vhdl-misc-ct3032n/`
- The Commit History (The Log Book)
  `https://github.com/errordeveloper/vhdl-misc-ct3032n/commits/`

References

The fallowing page contents were of great use for reference to VHDL methods:

- Dr J Plusquellic's Homepage, ECE, UNM
  `http://www.ece.unm.edu/~jimp/vhdl_fpgas/`