University of Huddersfield

VHDL Circuit Modelling and Simulation in Intel’s Quartus Prime software

Digital System Integration

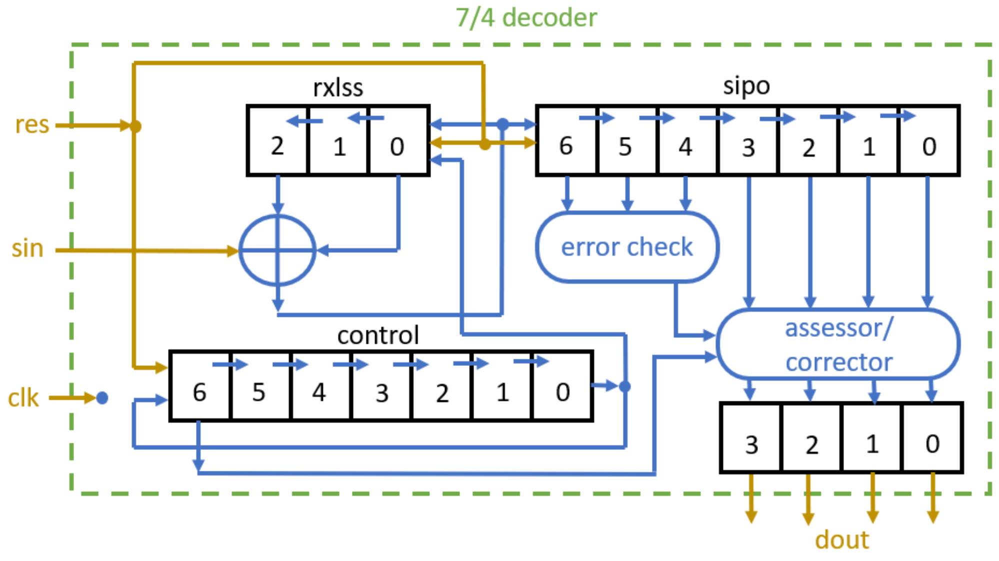
Jack Parkinson U1552044
11-11-2018

Contents

1. 7/4 Decoder

2.1. Design

Key	
🟩	Entity
🟧	Port
🟦	Sequential Logic (dependent on clk)
⬛	Register



The diagram above illustrates the operation of the 7/4 decoder which should be:

1. Can be reset at any time by setting the res port high
2. Decodes messages sent in serially on the sin port in packets of 7-bits
3. Outputs the 4-bits of data contained with the 7-bit message in parallel with any single bit errors corrected
4. Continues to operate once the 7-bit message has been decoded i.e. two or more 7-bit messages can be sent in one after another and be successfully decoded without having to reset the decoder

The purpose of each component illustrated in the diagram above is:

- **sin input port** – receives input signal
- **clk input port** - receives clock signal
- **res input port** - receives reset signal
- **dout output port** – transmits error corrected signal
- $\oplus$ **logic** – decodes input signal
- **control shift register** – keeps track of the number of bits decoded. When MSB is “1” it resets the rxlss register to “000” and it initiates the error check and assessor/corrector processes. Setting the rxlss register to “000” allows the next 7-bits to be correctly decoded
- **rxlss shift register** – stores the last 3 decoded bits to decode the input signal
- **sipo shift register** – stores the last 7 decoded bits (complete message) for the error check and assessor/corrector processes

The Boolean expression for the logic which decodes the input signal is:

$$sipo(6)\ \&\ rxlss(0) = sin \oplus rxlss(0) \oplus rxlss(2)$$

This gives the impulse response:

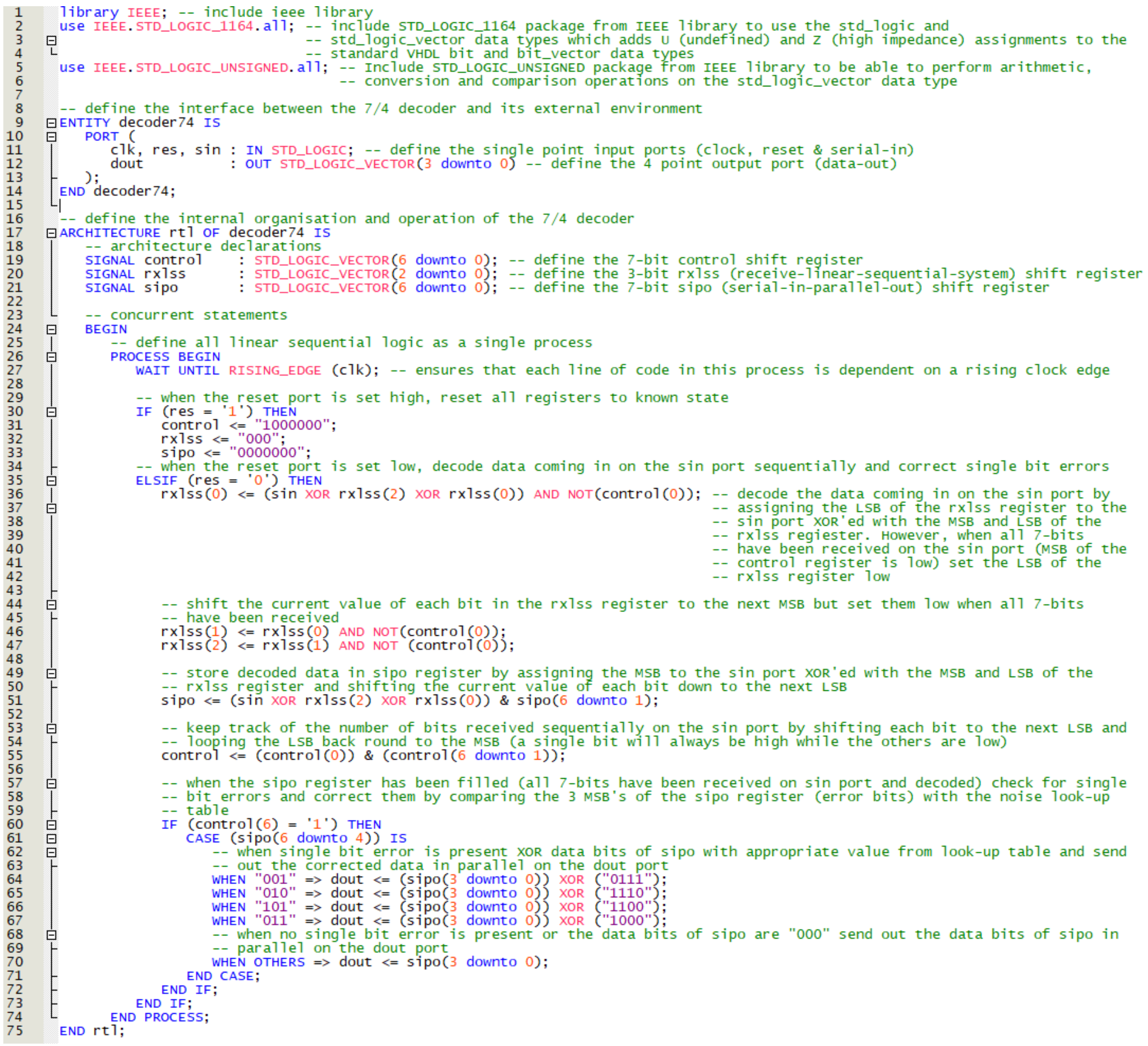
sin	$rxlss(0)$	$rxlss(1)$	$rxlss(2)$	$sipo(6)\ \&\ rxlss(0)$
1	0	0	0	1
0	1	0	0	1
0	1	1	0	1
0	1	1	1	0
0	0	1	1	1
0	1	0	1	0
0	0	1	0	0

From this impulse response, the Noise Look-Up table (**LUT**) can be given as:

sin	$sipo$
	6 5 4 3 2 1 0
0000001	0 0 1 0 1 1 1
0000010	0 1 0 1 1 1 0
0000100	1 0 1 1 1 0 0
0001000	0 1 1 1 0 0 0
0010000	1 1 1 0 0 0 0
0100000	1 1 0 0 0 0 0
1000000	1 0 0 0 0 0 0

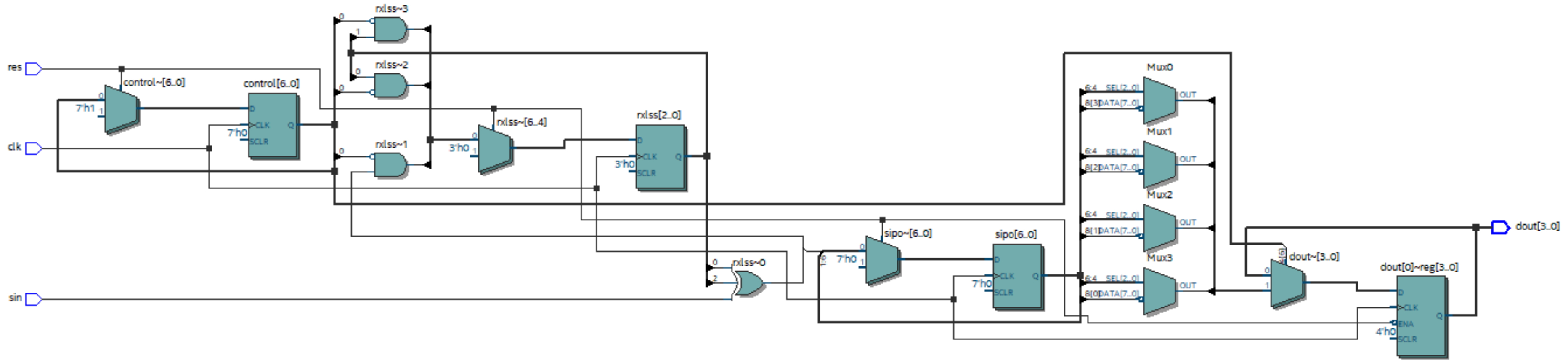
- error code
- correction code
- ignored (data = “0000”)

2.2. VHDL Code

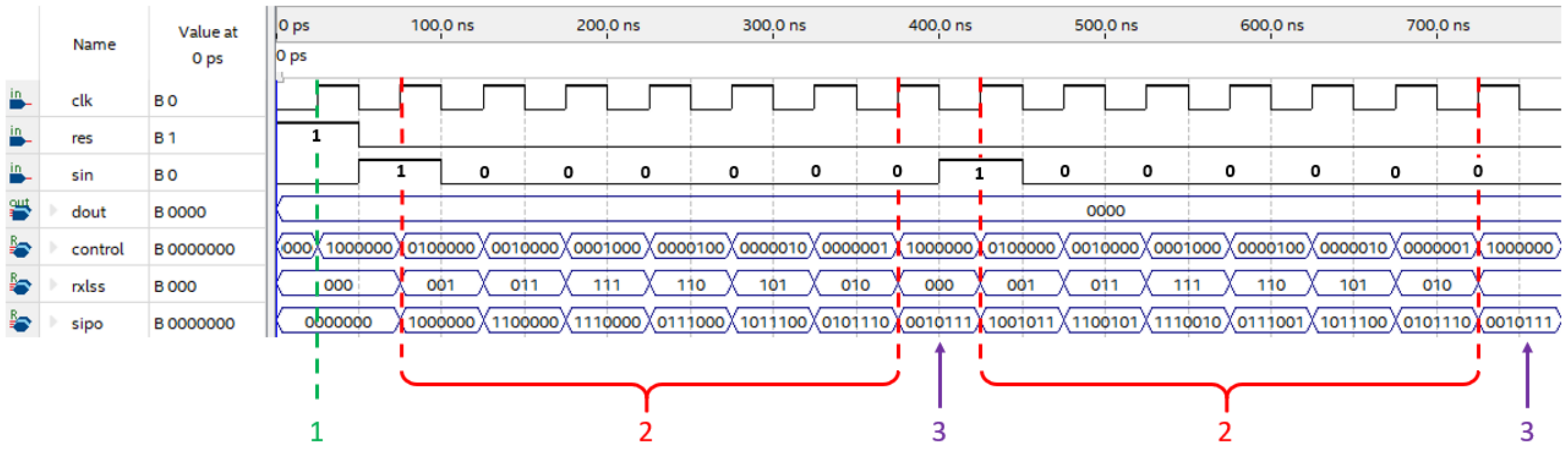


```vhdl
library IEEE; -- include ieee library
use IEEE.STD_LOGIC_1164.all; -- include STD_LOGIC_1164 package from IEEE library to use the std_logic and
                             -- std_logic_vector data types which adds U (undefined) and Z (high impedance) assignments to the
                             -- standard VHDL bit and bit_vector data types
use IEEE.STD_LOGIC_UNSIGNED.all; -- Include STD_LOGIC_UNSIGNED package from IEEE library to be able to perform arithmetic,
                                 -- conversion and comparison operations on the std_logic_vector data type

-- define the interface between the 7/4 decoder and its external environment
ENTITY decoder74 IS
   PORT (
      clk, res, sin : IN STD_LOGIC; -- define the single point input ports (clock, reset & serial-in)
      dout          : OUT STD_LOGIC_VECTOR(3 downto 0) -- define the 4 point output port (data-out)
   );
END decoder74;

-- define the internal organisation and operation of the 7/4 decoder
ARCHITECTURE rtl OF decoder74 IS
   -- architecture declarations
   SIGNAL control    : STD_LOGIC_VECTOR(6 downto 0); -- define the 7-bit control shift register
   SIGNAL rxlss      : STD_LOGIC_VECTOR(2 downto 0); -- define the 3-bit rxlss (receive-linear-sequential-system) shift register
   SIGNAL sipo       : STD_LOGIC_VECTOR(6 downto 0); -- define the 7-bit sipo (serial-in-parallel-out) shift register

   -- concurrent statements
   BEGIN
      -- define all linear sequential logic as a single process
      PROCESS BEGIN
         WAIT UNTIL RISING_EDGE (clk); -- ensures that each line of code in this process is dependent on a rising clock edge

         -- when the reset port is set high, reset all registers to known state
         IF (res = '1') THEN
            control <= "1000000";
            rxlss <= "000";
            sipo <= "0000000";
         -- when the reset port is set low, decode data coming in on the sin port sequentially and correct single bit errors
         ELSIF (res = '0') THEN
            rxlss(0) <= (sin XOR rxlss(2) XOR rxlss(0)) AND NOT(control(0)); -- decode the data coming in on the sin port by
                                                                             -- assigning the LSB of the rxlss register to the
                                                                             -- sin port XOR'ed with the MSB and LSB of the
                                                                             -- rxlss regiester. However, when all 7-bits
                                                                             -- have been received on the sin port (MSB of the
                                                                             -- control register is low) set the LSB of the
                                                                             -- rxlss register low

            -- shift the current value of each bit in the rxlss register to the next MSB but set them low when all 7-bits
            -- have been received
            rxlss(1) <= rxlss(0) AND NOT(control(0));
            rxlss(2) <= rxlss(1) AND NOT (control(0));

            -- store decoded data in sipo register by assigning the MSB to the sin port XOR'ed with the MSB and LSB of the
            -- rxlss register and shifting the current value of each bit down to the next LSB
            sipo <= (sin XOR rxlss(2) XOR rxlss(0)) & sipo(6 downto 1);

            -- keep track of the number of bits received sequentially on the sin port by shifting each bit to the next LSB and
            -- looping the LSB back round to the MSB (a single bit will always be high while the others are low)
            control <= (control(0)) & (control(6 downto 1));

            -- when the sipo register has been filled (all 7-bits have been received on sin port and decoded) check for single
            -- bit errors and correct them by comparing the 3 MSB's of the sipo register (error bits) with the noise look-up
            -- table
            IF (control(6) = '1') THEN
               CASE (sipo(6 downto 4)) IS
                  -- when single bit error is present XOR data bits of sipo with appropriate value from look-up table and send
                  -- out the corrected data in parallel on the dout port
                  WHEN "001" => dout <= (sipo(3 downto 0)) XOR ("0111");
                  WHEN "010" => dout <= (sipo(3 downto 0)) XOR ("1110");
                  WHEN "101" => dout <= (sipo(3 downto 0)) XOR ("1100");
                  WHEN "011" => dout <= (sipo(3 downto 0)) XOR ("1000");
                  -- when no single bit error is present or the data bits of sipo are "000" send out the data bits of sipo in
                  -- parallel on the dout port
                  WHEN OTHERS => dout <= sipo(3 downto 0);
               END CASE;
            END IF;
         END IF;
      END PROCESS;
END rtl;
```

Compiled to schematic



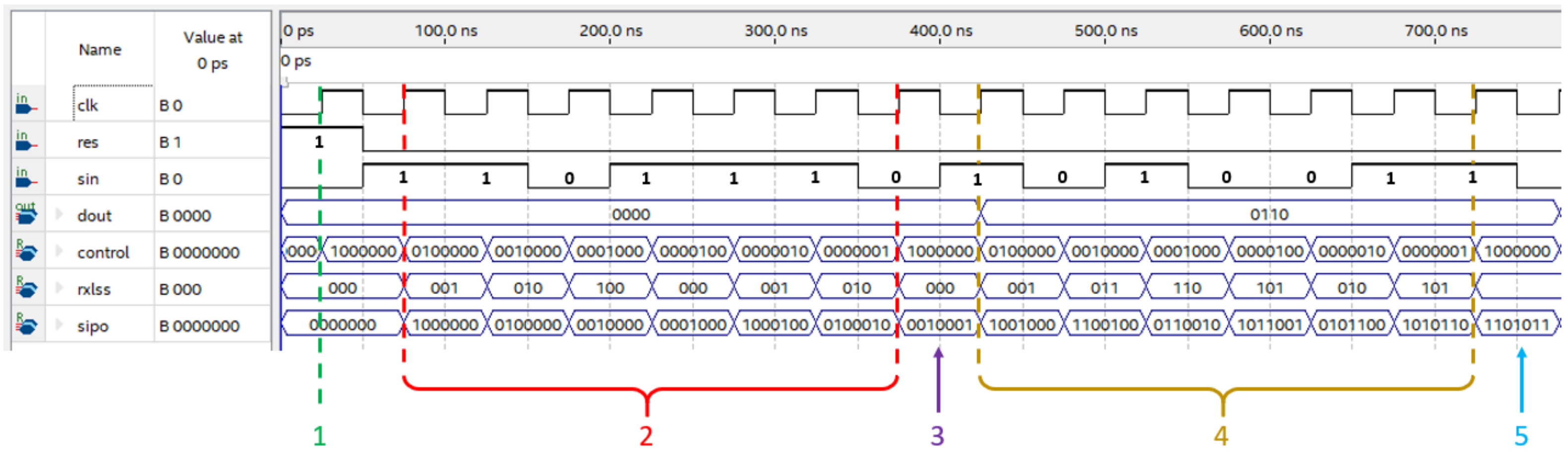
2.3. ModelSim Waveform



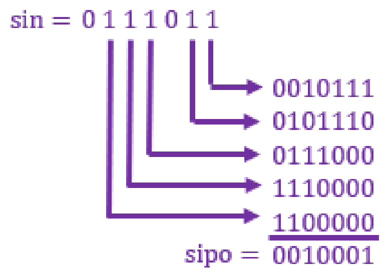
1. Reset registers to known state
2. Serially load in "0000001"
3. After 7-bits have been decoded (control = "1000000"), sipo = "0010111" which matches the impulse response on page 3

dout = "0000" throughout as sipo contains the error code "001" from the LUT which means the data bits from sipo (four LSB's) are XOR'ed with the corresponding correction code from the LUT

$$\begin{array}{ccccc} sipo & & code & & dout \\ 0111 & \oplus & 0111 & = & 0000 \end{array}$$



1. Reset registers to known state
2. Serially load in "0111011"
3. After 7-bits have been decoded (control = "1000000"), sipo = "0010001" to check if this is correct, XOR corresponding rows from LUT

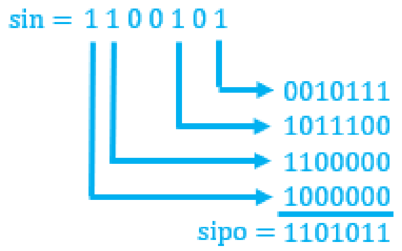


4. Serially load in "1100101"

When control = "1000000", sipo = "0010001" which contains the error code "001" from LUT. To work out what dout for the first 7-bit message load in during step 2, the data bits from sipo need to be XOR'ed with the corresponding correction code from the LUT

$$\begin{array}{ccc} sipo & code & dout \\ 0001 \oplus & 0111 = & 0110 \end{array}$$

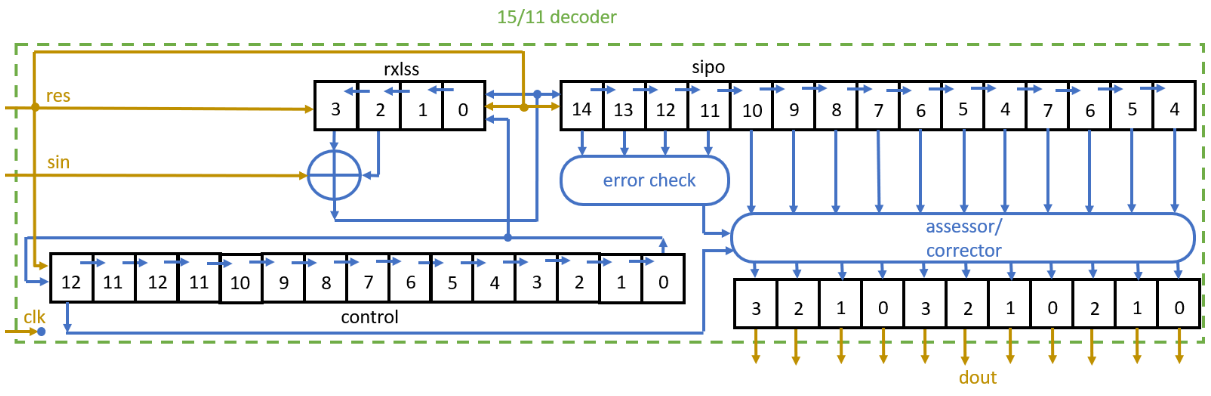
5. After 7-bits have been decoded (control = “1000000”), sipo = “1101011” to check if this is correct, XOR corresponding rows from LUT



2. 15/11 Decoder

3.1. Design

Key	
🟩	Entity
🟧	Port
🟦	Sequential Logic (dependent on clk)
⬛	Register



The diagram above illustrates the operation of the 15/11 decoder which should be:

1. Can be reset at any time by setting the res port high
2. Decodes messages sent in serially on the sin port in packets of 15-bits
3. Outputs the 11-bits of data contained with the 15-bit message in parallel with any single bit errors corrected
4. Continues to operate once the 15-bit message has been decoded i.e. two or more 15-bit messages can be sent in one after another and be successfully decoded without having to reset the decoder

The purpose of each component illustrated in the diagram above is:

- **sin input port** – receives input signal
- **clk input port** - receives clock signal
- **res input port** - receives reset signal
- **dout output port** – transmits error corrected signal
- $\oplus$ **logic** – decodes input signal
- **control shift register** – keeps track of the number of bits decoded. When MSB is “1” it resets the rxlss register to “0000” and it initiates the error check and assessor/corrector processes. Setting the rxlss register to “0000” allows the next 15-bits to be correctly decoded
- **rxlss shift register** – stores the last 4 decoded bits to decode the input signal
- **sipo shift register** – stores the last 15 decoded bits (complete message) for the error check and assessor/corrector processes

The Boolean expression for the logic which decodes the input signal is:

$$sipo(14)\,\&\,rxlss(0) = sin \oplus rxlss(2) \oplus rxlss(3)$$

This gives the impulse response:

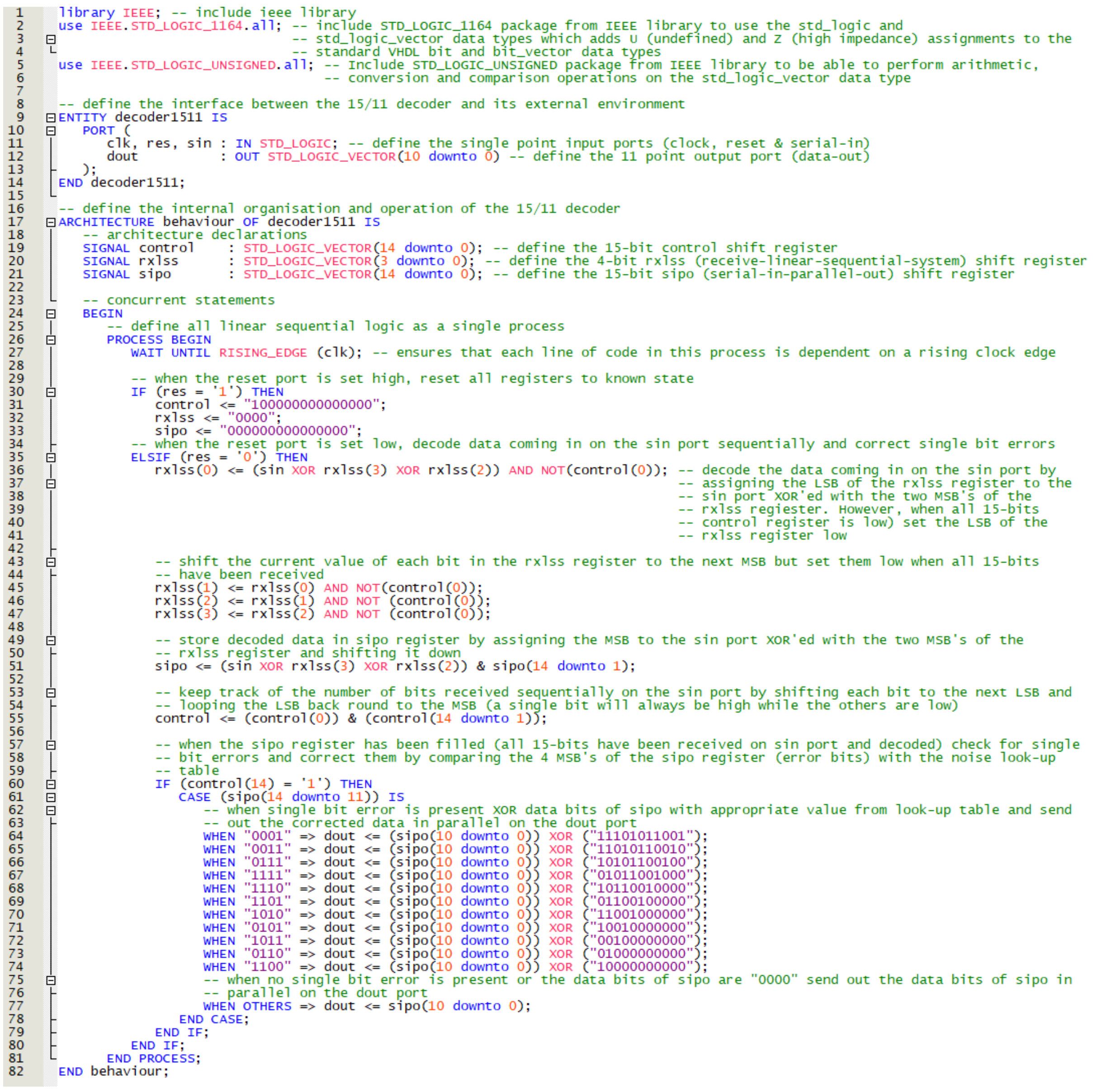
sin	$rxlss(0)$	$rxlss(1)$	$rxlss(2)$	$rxlss(3)$	$sipo(14)\,\&\,rxlss(0)$
1	0	0	0	0	1
0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	1
0	1	0	0	1	1
0	1	1	0	0	0
0	0	1	1	0	1
0	1	0	1	1	0
0	0	1	0	1	1
0	1	0	1	0	1
0	1	1	0	1	1
0	1	1	1	0	1
0	1	1	1	1	0
0	0	1	1	1	0
0	0	0	1	1	0

From this impulse response, the Noise Look-Up table (**LUT**) can be given as:

sin	$sipo$														
	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
000000000000001	0	0	0	1	1	1	1	0	1	0	1	1	0	0	1
000000000000010	0	0	1	1	1	1	0	1	0	1	1	0	0	1	0
000000000000100	0	1	1	1	1	0	1	0	1	1	0	0	1	0	0
000000000001000	1	1	1	1	0	1	0	1	1	0	0	1	0	0	0
000000000010000	1	1	1	0	1	0	1	1	0	0	1	0	0	0	0
000000000100000	1	1	0	1	0	1	1	0	0	1	0	0	0	0	0
000000001000000	1	0	1	0	1	1	0	0	1	0	0	0	0	0	0
000000010000000	0	1	0	1	1	0	0	1	0	0	0	0	0	0	0
000000100000000	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0
000001000000000	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0
000010000000000	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0
000100000000000	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
001000000000000	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
010000000000000	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
100000000000000	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

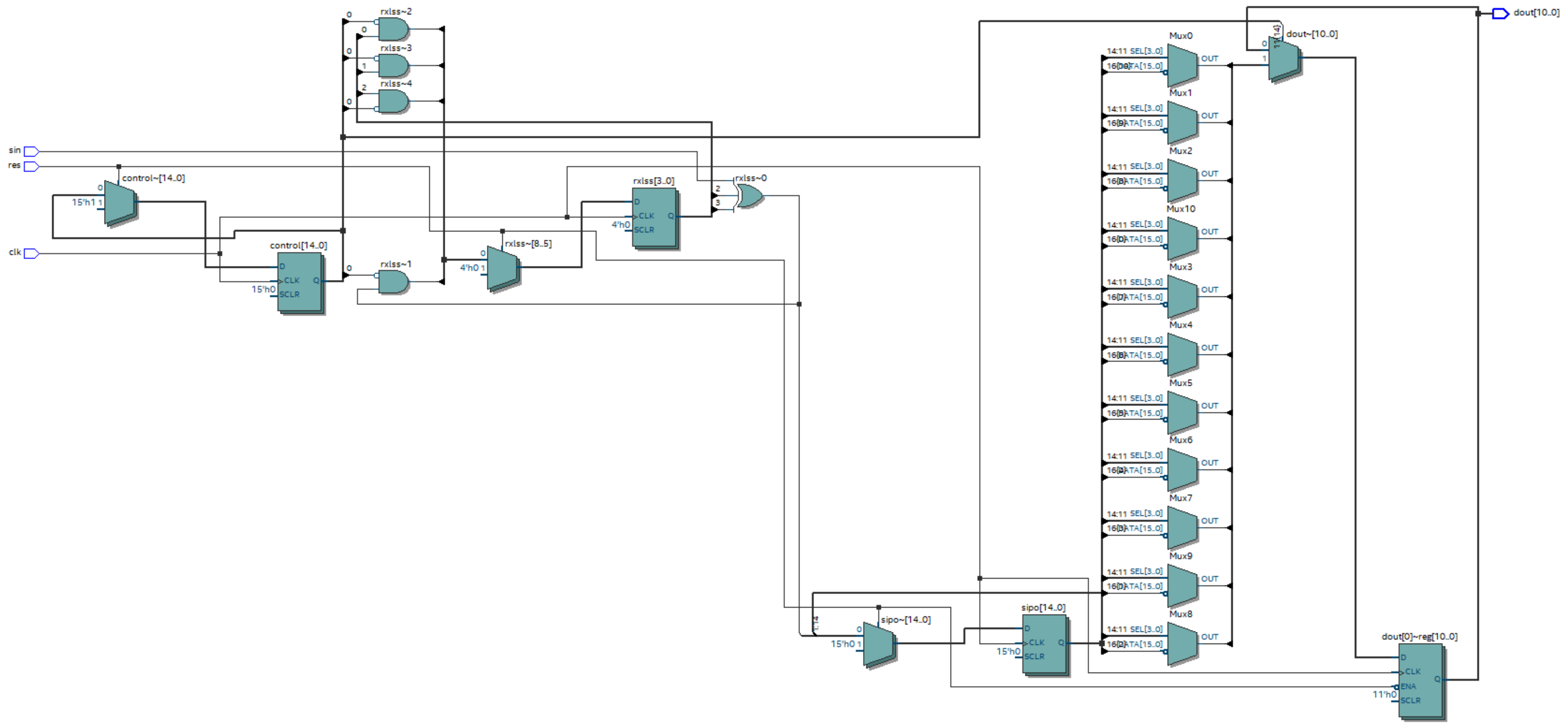
- error code
- correction code
- ignored (data = “00000000000”)

3.2. VHDL Code

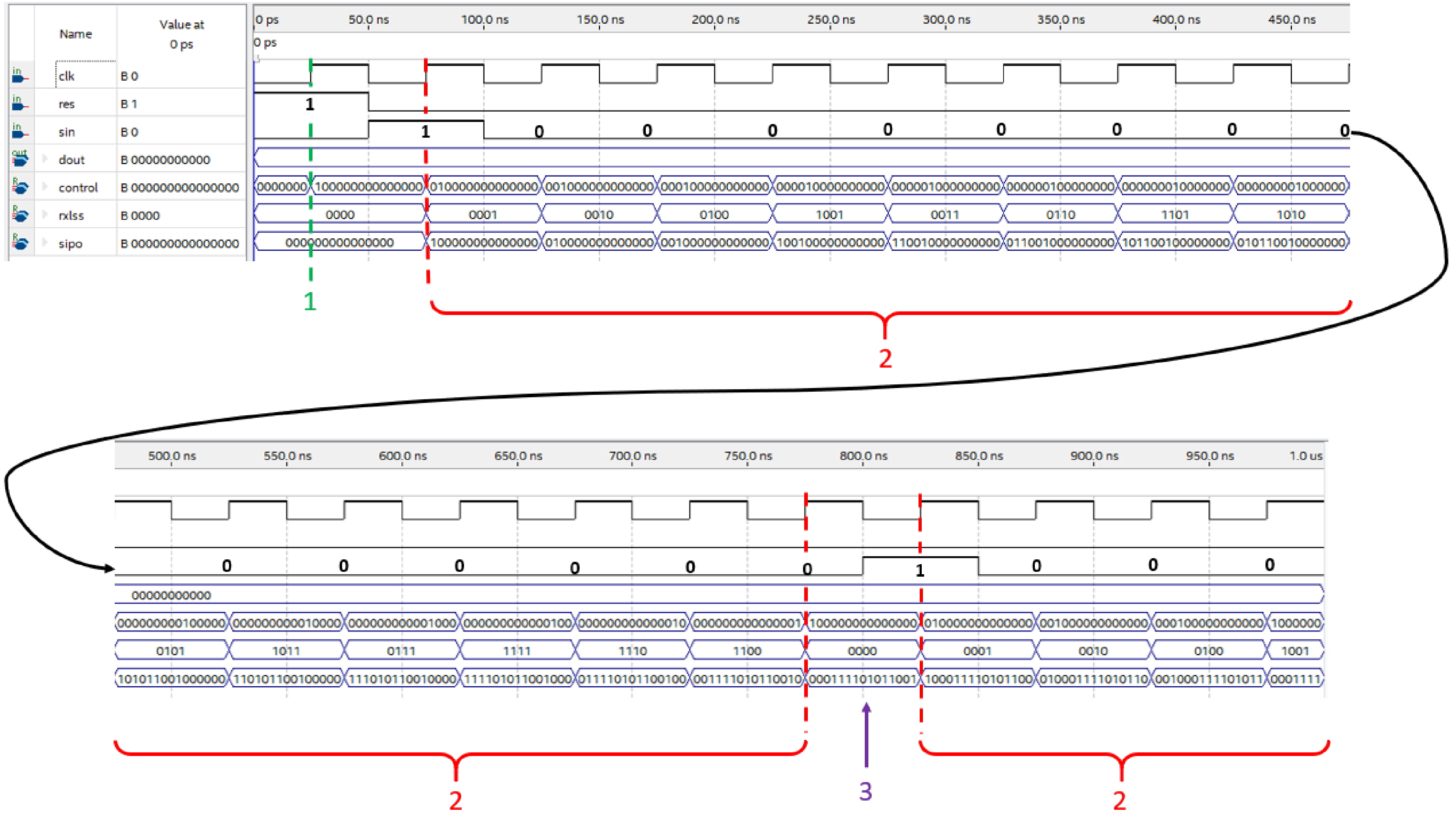


```vhdl
library IEEE; -- include ieee library
use IEEE.STD_LOGIC_1164.all; -- include STD_LOGIC_1164 package from IEEE library to use the std_logic and
                             -- std_logic_vector data types which adds U (undefined) and Z (high impedance) assignments to the
                             -- standard VHDL bit and bit_vector data types
use IEEE.STD_LOGIC_UNSIGNED.all; -- Include STD_LOGIC_UNSIGNED package from IEEE library to be able to perform arithmetic,
                                 -- conversion and comparison operations on the std_logic_vector data type

-- define the interface between the 15/11 decoder and its external environment
ENTITY decoder1511 IS
   PORT (
      clk, res, sin : IN STD_LOGIC; -- define the single point input ports (clock, reset & serial-in)
      dout          : OUT STD_LOGIC_VECTOR(10 downto 0) -- define the 11 point output port (data-out)
   );
END decoder1511;

-- define the internal organisation and operation of the 15/11 decoder
ARCHITECTURE behaviour OF decoder1511 IS
   -- architecture declarations
   SIGNAL control     : STD_LOGIC_VECTOR(14 downto 0); -- define the 15-bit control shift register
   SIGNAL rxlss       : STD_LOGIC_VECTOR(3 downto 0); -- define the 4-bit rxlss (receive-linear-sequential-system) shift register
   SIGNAL sipo        : STD_LOGIC_VECTOR(14 downto 0); -- define the 15-bit sipo (serial-in-parallel-out) shift register

   -- concurrent statements
   BEGIN
      -- define all linear sequential logic as a single process
      PROCESS BEGIN
         WAIT UNTIL RISING_EDGE (clk); -- ensures that each line of code in this process is dependent on a rising clock edge

         -- when the reset port is set high, reset all registers to known state
         IF (res = '1') THEN
            control <= "100000000000000";
            rxlss <= "0000";
            sipo <= "000000000000000";
         -- when the reset port is set low, decode data coming in on the sin port sequentially and correct single bit errors
         ELSIF (res = '0') THEN
            rxlss(0) <= (sin XOR rxlss(3) XOR rxlss(2)) AND NOT(control(0)); -- decode the data coming in on the sin port by
                                                                             -- assigning the LSB of the rxlss register to the
                                                                             -- sin port XOR'ed with the two MSB's of the
                                                                             -- rxlss regiester. However, when all 15-bits
                                                                             -- control register is low) set the LSB of the
                                                                             -- rxlss register low

            -- shift the current value of each bit in the rxlss register to the next MSB but set them low when all 15-bits
            -- have been received
            rxlss(1) <= rxlss(0) AND NOT(control(0));
            rxlss(2) <= rxlss(1) AND NOT (control(0));
            rxlss(3) <= rxlss(2) AND NOT (control(0));

            -- store decoded data in sipo register by assigning the MSB to the sin port XOR'ed with the two MSB's of the
            -- rxlss register and shifting it down
            sipo <= (sin XOR rxlss(3) XOR rxlss(2)) & sipo(14 downto 1);

            -- keep track of the number of bits received sequentially on the sin port by shifting each bit to the next LSB and
            -- looping the LSB back round to the MSB (a single bit will always be high while the others are low)
            control <= (control(0)) & (control(14 downto 1));

            -- when the sipo register has been filled (all 15-bits have been received on sin port and decoded) check for single
            -- bit errors and correct them by comparing the 4 MSB's of the sipo register (error bits) with the noise look-up
            -- table
            IF (control(14) = '1') THEN
               CASE (sipo(14 downto 11)) IS
                  -- when single bit error is present XOR data bits of sipo with appropriate value from look-up table and send
                  -- out the corrected data in parallel on the dout port
                  WHEN "0001" => dout <= (sipo(10 downto 0)) XOR ("11101011001");
                  WHEN "0011" => dout <= (sipo(10 downto 0)) XOR ("11010110010");
                  WHEN "0111" => dout <= (sipo(10 downto 0)) XOR ("10101100100");
                  WHEN "1111" => dout <= (sipo(10 downto 0)) XOR ("01011001000");
                  WHEN "1110" => dout <= (sipo(10 downto 0)) XOR ("10110010000");
                  WHEN "1101" => dout <= (sipo(10 downto 0)) XOR ("01100100000");
                  WHEN "1010" => dout <= (sipo(10 downto 0)) XOR ("11001000000");
                  WHEN "0101" => dout <= (sipo(10 downto 0)) XOR ("10010000000");
                  WHEN "1011" => dout <= (sipo(10 downto 0)) XOR ("00100000000");
                  WHEN "0110" => dout <= (sipo(10 downto 0)) XOR ("01000000000");
                  WHEN "1100" => dout <= (sipo(10 downto 0)) XOR ("10000000000");
                  -- when no single bit error is present or the data bits of sipo are "0000" send out the data bits of sipo in
                  -- parallel on the dout port
                  WHEN OTHERS => dout <= sipo(10 downto 0);
               END CASE;
            END IF;
         END IF;
      END PROCESS;
END behaviour;
```

Compiled to schematic



3.3. ModelSim Waveform

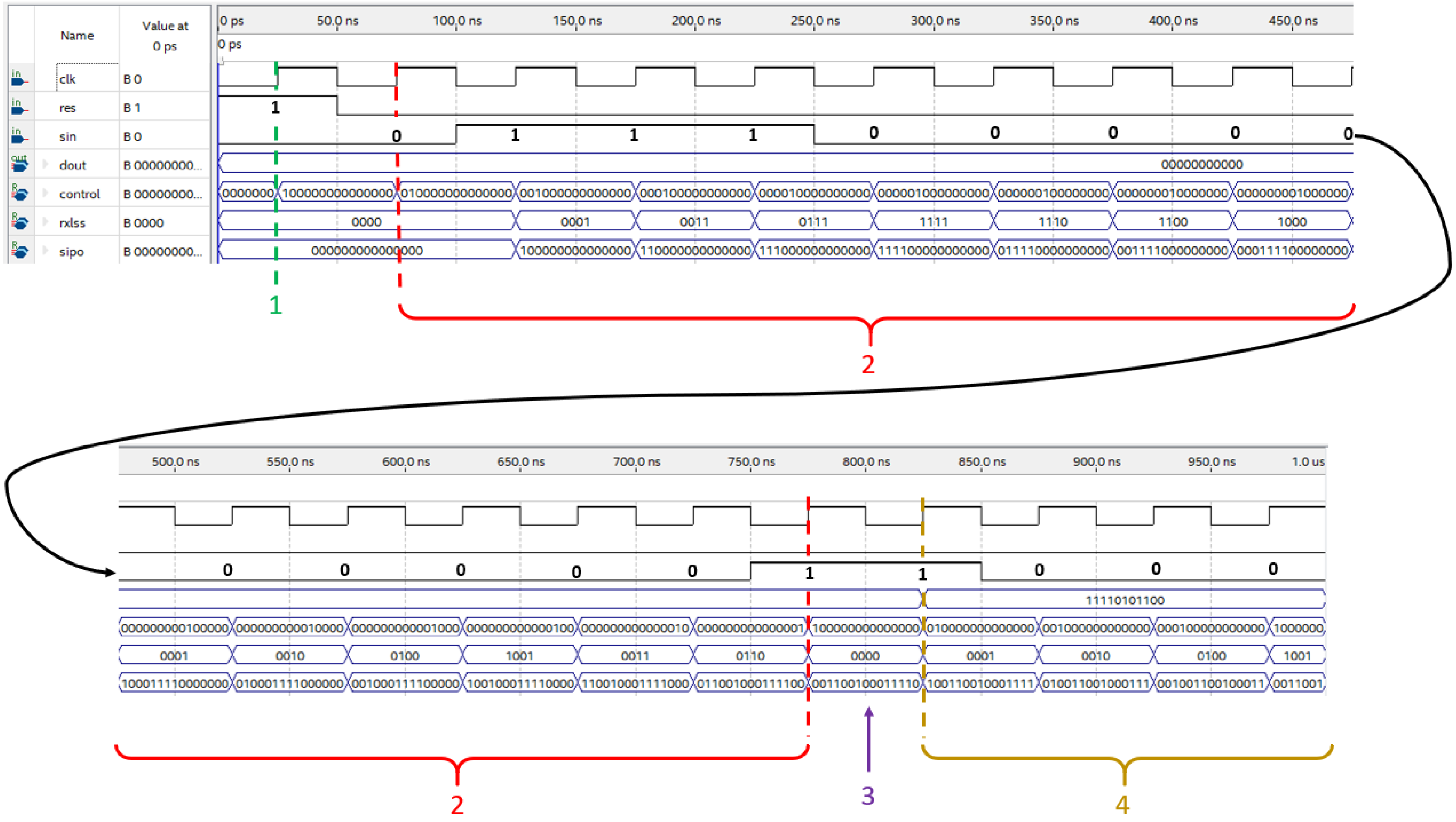


1. Reset registers to known state
2. Serially load in “000000000000001”

3. After 15-bits have been decoded (control = “100000000000000”), sipo = “000111101011001” which matches the impulse response on page 10

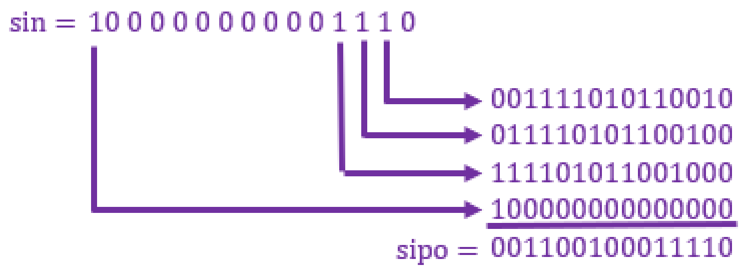
   dout = “00000000000” throughout as sipo contains the error code “0001” from the LUT which means the data bits from sipo (eleven LSB’s) are XOR’ed with the corresponding correction code from the LUT

$$\underset{sipo}{11101011001} \oplus \underset{code}{11101011001} = \underset{dout}{00000000000}$$



1. Reset registers to known state
2. Serially load in "100000000001110"

3. After 15-bits have been decoded (control = “100000000000000”), sipo = “001100100011110” to check if this is correct, XOR corresponding rows from LUT



4. Serially load in “000000000000001”

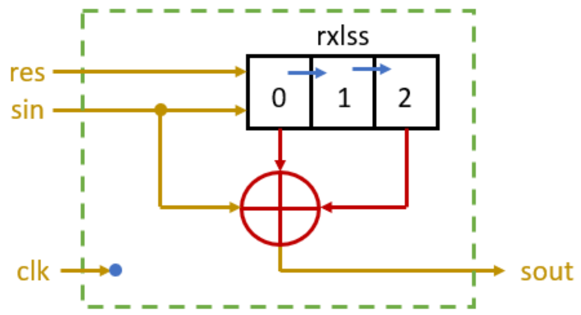
   When control = “100000000000000”, sipo = “001100100011110” which contains the error code “0011” from LUT. To work out what dout for the 15-bit message load in during step 2, the data bits from sipo need to be XOR’ed with the corresponding correction code from the LUT

$$\overset{sipo}{00100011110} \oplus \overset{code}{11010110010} = \overset{dout}{11110101100}$$

3. 7/4 Encoder

4.1. Design

Key	
🟩	Entity
🟨	Port
🟦	Sequential Logic (dependent on clk)
⬛	Register
🟥	Combination Logic



The diagram above illustrates the operation of the 7/4 encoder which should be:

1. Can be reset at any time by setting the res port high
2. Encodes messages sent in serially on the sin port in packets of 7-bits
3. Outputs the encoded messages serially
4. Continues to operate once the 7-bit message has been encoded i.e. two or more 7-bit messages can be sent in one after another and be successfully encoded without having to reset the encoder

The purpose of each component illustrated in the diagram above is:

- **sin input port** – receives input signal
- **clk input port** - receives clock signal
- **res input port** - receives reset signal
- **sout output port** – transmits encoded signal
- $\oplus$ **logic** – encodes input signal
- **rxlss shift register** – stores the last 3 bits input to encode the input signal

The Boolean expression for the logic which encodes the input signal is:

$$sout = sin \oplus rxlss(0) \oplus rxlss(2)$$

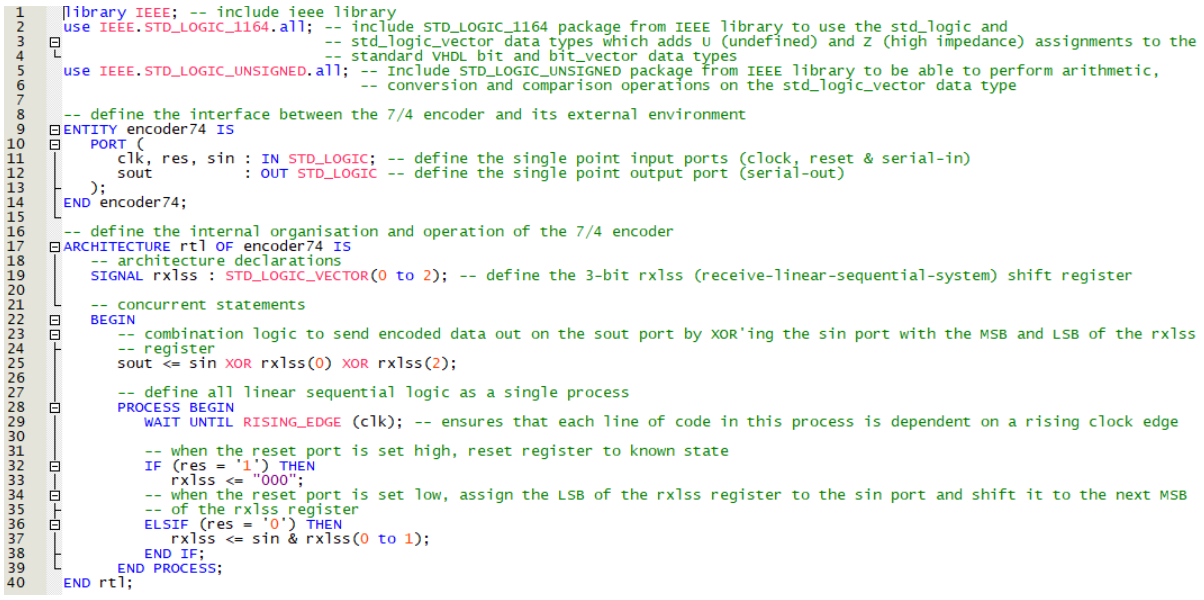
This gives the impulse response:

sin	$rxlss(0)$	$rxlss(1)$	$rxlss(2)$	$sout$
1	0	0	0	1
0	1	0	0	1
0	0	1	0	0
0	0	0	1	1
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

From this impulse response, the Noise Look-Up table (**LUT**) can be given as:

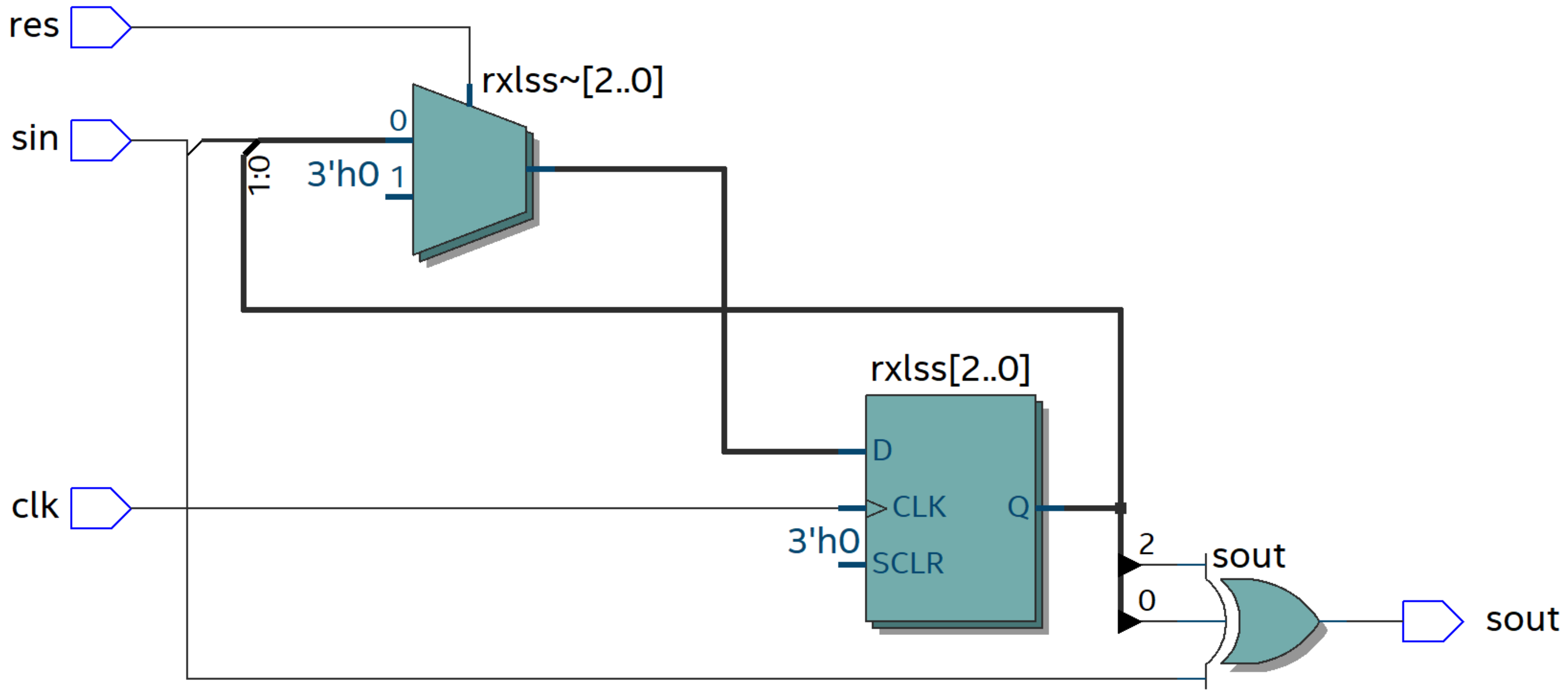
sin	$sout$
0000001	0001011
0000010	0010110
0000100	0101100
0001000	1011000
0010000	0110000
0100000	1100000
1000000	1000000

4.2. VHDL Code

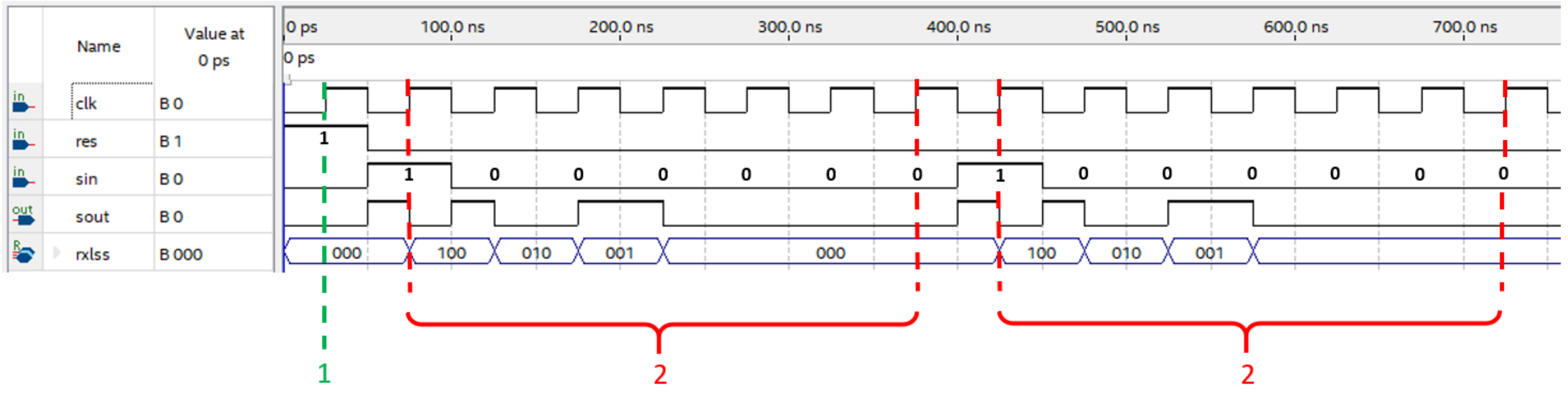


```vhdl
library IEEE; -- include ieee library
use IEEE.STD_LOGIC_1164.all; -- include STD_LOGIC_1164 package from IEEE library to use the std_logic and
                             -- std_logic_vector data types which adds U (undefined) and Z (high impedance) assignments to the
                             -- standard VHDL bit and bit_vector data types
use IEEE.STD_LOGIC_UNSIGNED.all; -- Include STD_LOGIC_UNSIGNED package from IEEE library to be able to perform arithmetic,
                                 -- conversion and comparison operations on the std_logic_vector data type

-- define the interface between the 7/4 encoder and its external environment
ENTITY encoder74 IS
   PORT (
      clk, res, sin : IN STD_LOGIC; -- define the single point input ports (clock, reset & serial-in)
      sout          : OUT STD_LOGIC -- define the single point output port (serial-out)
   );
END encoder74;

-- define the internal organisation and operation of the 7/4 encoder
ARCHITECTURE rtl OF encoder74 IS
   -- architecture declarations
   SIGNAL rxlss : STD_LOGIC_VECTOR(0 to 2); -- define the 3-bit rxlss (receive-linear-sequential-system) shift register

   -- concurrent statements
   BEGIN
      -- combination logic to send encoded data out on the sout port by XOR'ing the sin port with the MSB and LSB of the rxlss
      -- register
      sout <= sin XOR rxlss(0) XOR rxlss(2);

      -- define all linear sequential logic as a single process
      PROCESS BEGIN
         WAIT UNTIL RISING_EDGE (clk); -- ensures that each line of code in this process is dependent on a rising clock edge

         -- when the reset port is set high, reset register to known state
         IF (res = '1') THEN
            rxlss <= "000";
         -- when the reset port is set low, assign the LSB of the rxlss register to the sin port and shift it to the next MSB
         -- of the rxlss register
         ELSIF (res = '0') THEN
            rxlss <= sin & rxlss(0 to 1);
         END IF;
      END PROCESS;
END rtl;
```

Compiled to schematic

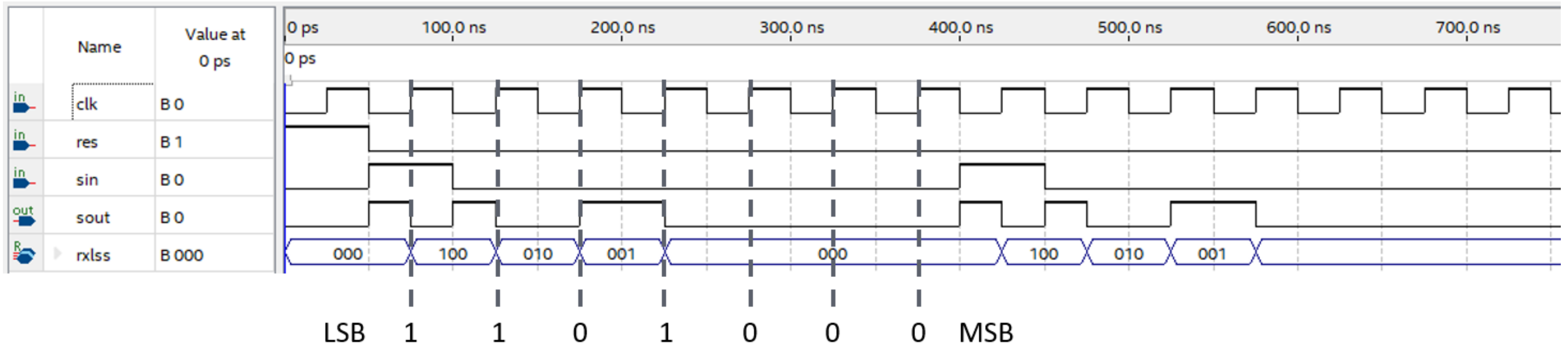


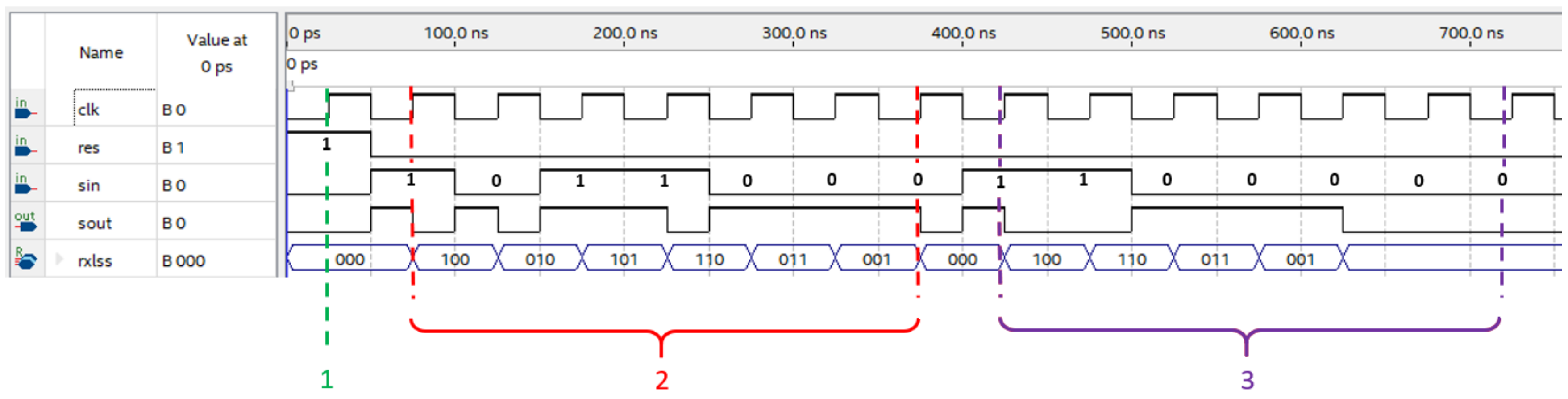
4.3. ModelSim Waveform



1. Reset registers to known state
2. Serially load in “0000001” and send encoded signal out

At first glance, sout does not appear to match that of the impulse response on page 18 due to the encoding logic being implemented as combination logic rather than sequential logic so it is not depend on the clock. Due to the delay in the combination logic, the decoder will read falling edges on sout as “1” and rising edges as “0” hence sout does match the impulse response “0001011”





1. Reset registers to known state
3. Serially load in "0001101" and send encoded signal out

   To check what sout should be, XOR corresponding rows from LUT

   sin = 0 0 0 1 1 0 1

   0001011
   0101100
   1011000

   sout = 1111111

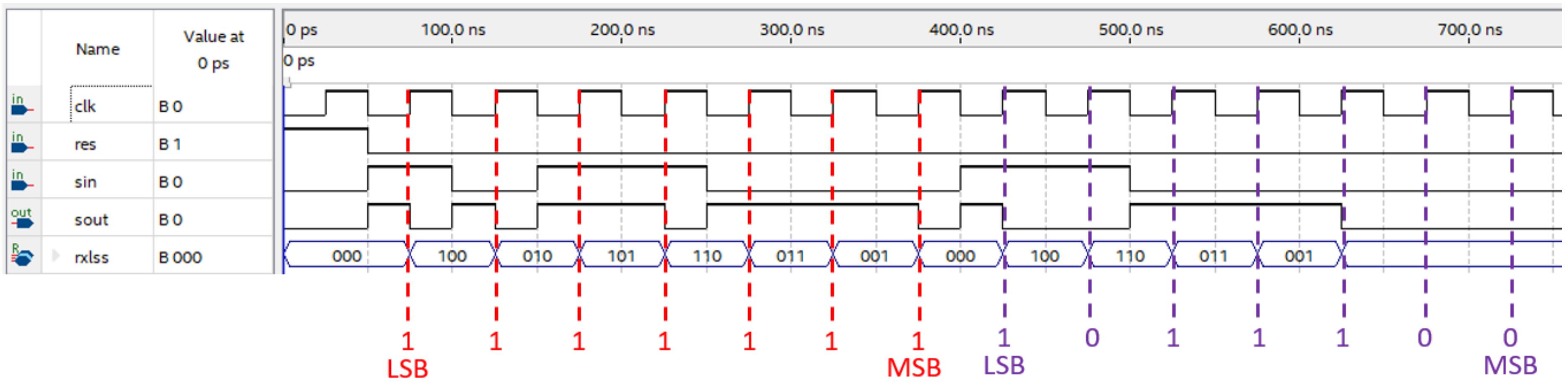
4. Serially load in "0000011" and send encoded signal out

   To check what sout should be, XOR corresponding rows from LUT

   sin = 0 0 0 0 0 1 1

   0001011
   0010110

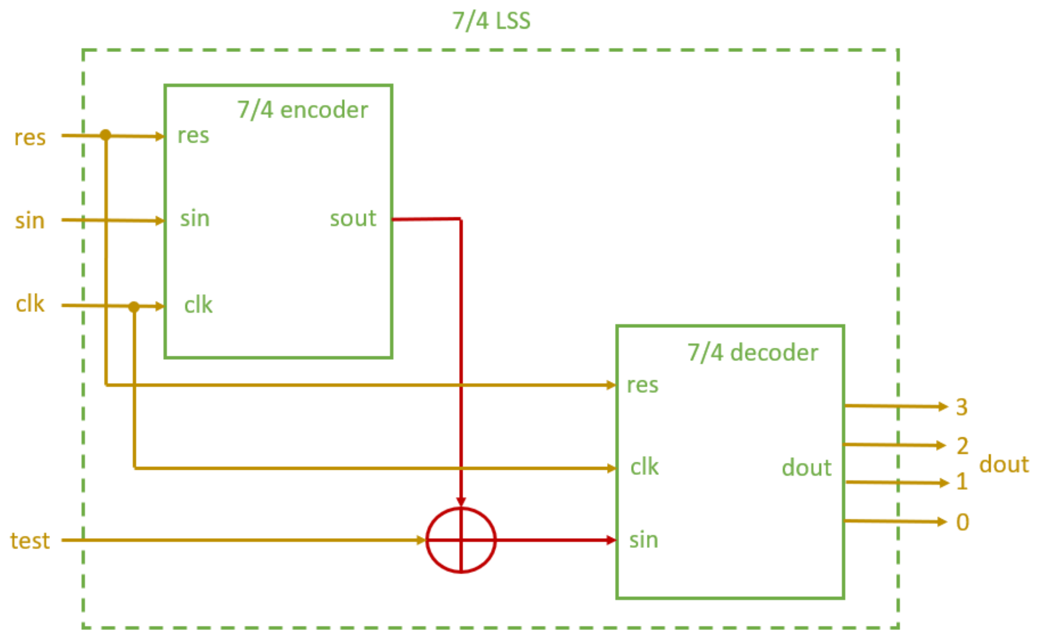
   sout = 0011101


Name
Value at 0 ps
0 ps
100.0 ns
200.0 ns
300.0 ns
400.0 ns
500.0 ns
600.0 ns
700.0 ns
clk B 0
res B 1
sin B 0
sout B 0
rxlss B 000
000
100
010
101
110
011
001
000
100
110
011
001
1 1 1 1 1 1 1
LSB
MSB
1 0 1 1 1 0 0
LSB
MSB

4. 7/4 Linear Sequential Coding System

5.1. Design

Key	
🟩	Entity
🟨	Port
🟥	Combination Logic



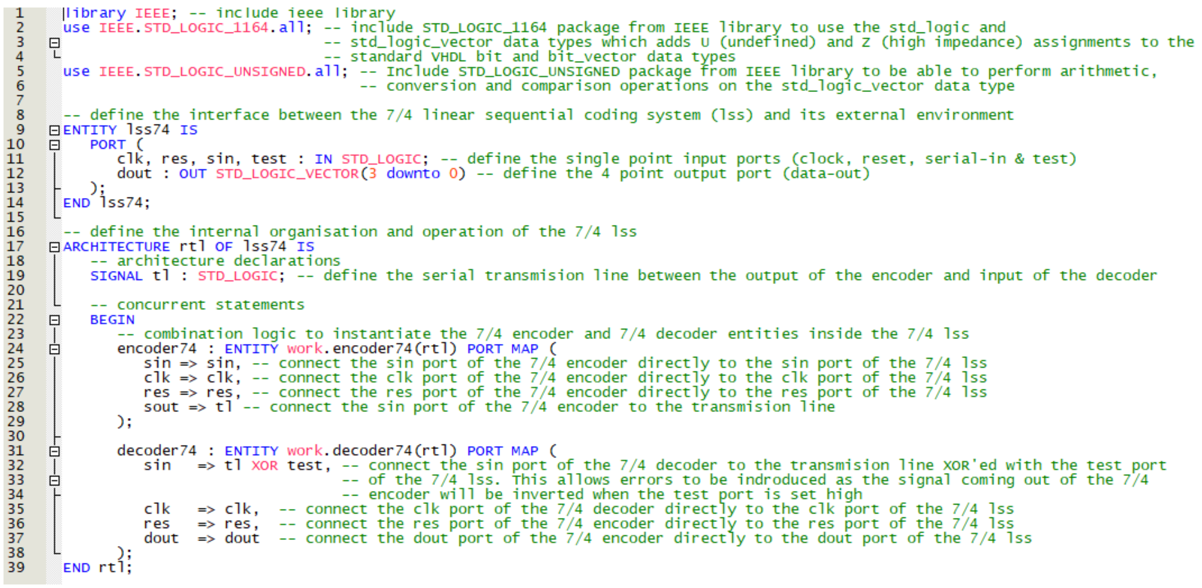
The diagram above illustrates the operation of the 7/4 LSS which should be:

1. Can be reset at any time by setting the res port high
2. Encodes and decodes messages sent in serially on the sin port in packets of 7-bits such that any single bit errors introduced in the transmission line between the encoder and decoder are corrected
3. Outputs the 4-bits of data contained with the 7-bit message in parallel
4. Continues to operate once the 7-bit message has been encoded and decoded i.e. two or more 7-bit messages can be sent in one after another and be successfully encoded and decoded without having to reset the LSS

The purpose of each component illustrated in the diagram above is:

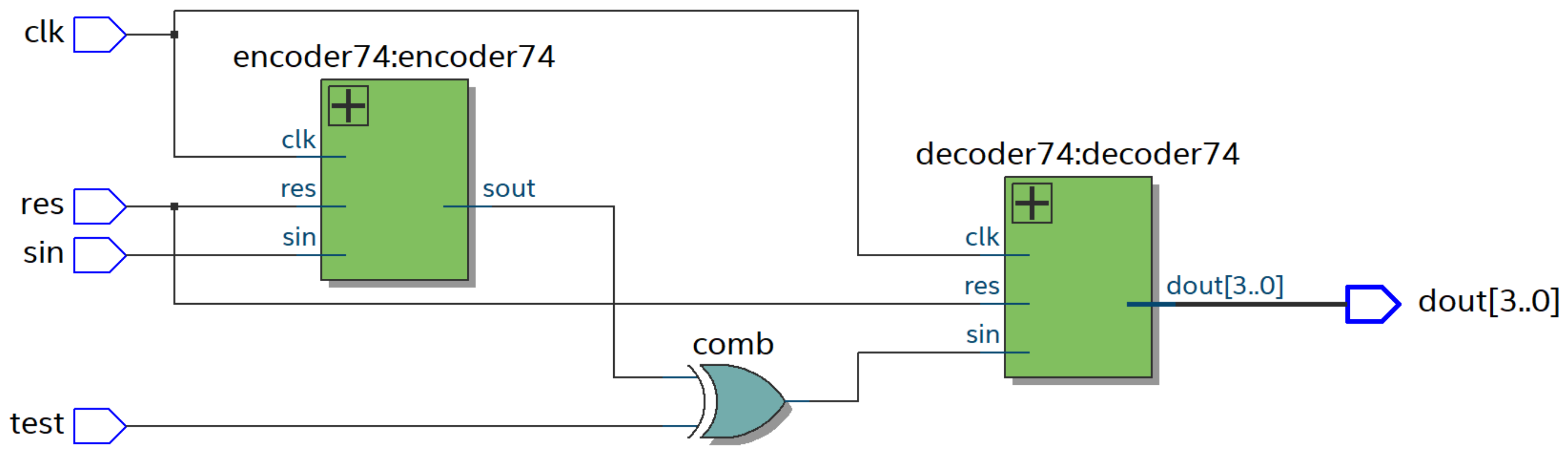
- **sin input port** – receives input signal
- **clk input port** - receives clock signal
- **res input port** - receives reset signal
- **test input port & $\oplus$ logic** – inverts the signal present on transmission line between the encoder and decoder to introduce errors
- **dout output port** – – transmits error corrected signal

5.2. VHDL Code

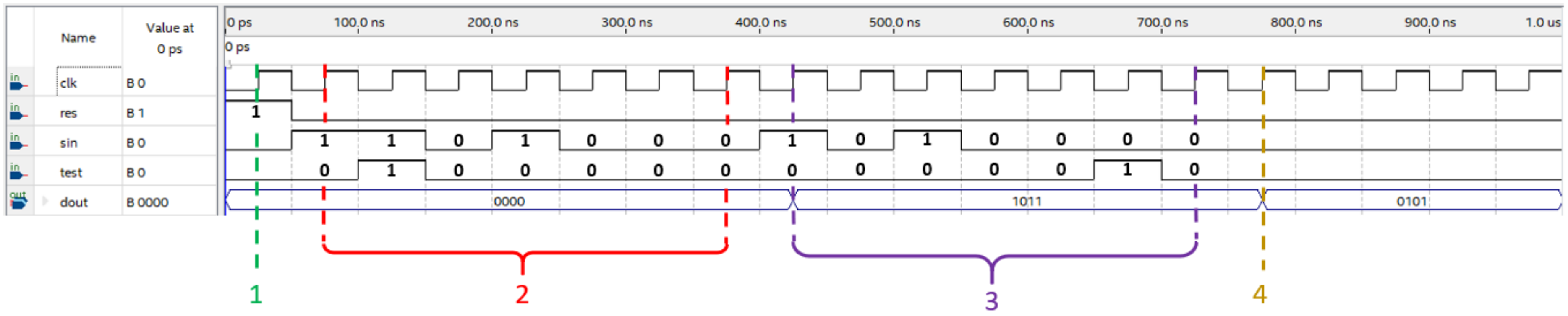


```vhdl
library IEEE; -- include ieee library
use IEEE.STD_LOGIC_1164.all; -- include STD_LOGIC_1164 package from IEEE library to use the std_logic and
                             -- std_logic_vector data types which adds U (undefined) and Z (high impedance) assignments to the
                             -- standard VHDL bit and bit_vector data types
use IEEE.STD_LOGIC_UNSIGNED.all; -- Include STD_LOGIC_UNSIGNED package from IEEE library to be able to perform arithmetic,
                                 -- conversion and comparison operations on the std_logic_vector data type

-- define the interface between the 7/4 linear sequential coding system (lss) and its external environment
ENTITY lss74 IS
   PORT (
      clk, res, sin, test : IN STD_LOGIC; -- define the single point input ports (clock, reset, serial-in & test)
      dout : OUT STD_LOGIC_VECTOR(3 downto 0) -- define the 4 point output port (data-out)
   );
END lss74;

-- define the internal organisation and operation of the 7/4 lss
ARCHITECTURE rtl OF lss74 IS
   -- architecture declarations
   SIGNAL tl : STD_LOGIC; -- define the serial transmision line between the output of the encoder and input of the decoder

   -- concurrent statements
   BEGIN
      -- combination logic to instantiate the 7/4 encoder and 7/4 decoder entities inside the 7/4 lss
      encoder74 : ENTITY work.encoder74(rtl) PORT MAP (
         sin => sin, -- connect the sin port of the 7/4 encoder directly to the sin port of the 7/4 lss
         clk => clk, -- connect the clk port of the 7/4 encoder directly to the clk port of the 7/4 lss
         res => res, -- connect the res port of the 7/4 encoder directly to the res port of the 7/4 lss
         sout => tl -- connect the sin port of the 7/4 encoder to the transmision line
      );

      decoder74 : ENTITY work.decoder74(rtl) PORT MAP (
         sin   => tl XOR test, -- connect the sin port of the 7/4 decoder to the transmision line XOR'ed with the test port
                               -- of the 7/4 lss. This allows errors to be indroduced as the signal coming out of the 7/4
                               -- encoder will be inverted when the test port is set high
         clk   => clk,  -- connect the clk port of the 7/4 decoder directly to the clk port of the 7/4 lss
         res   => res,  -- connect the res port of the 7/4 encoder directly to the res port of the 7/4 lss
         dout  => dout  -- connect the dout port of the 7/4 encoder directly to the dout port of the 7/4 lss
      );
END rtl;
```

Compiled to schematic



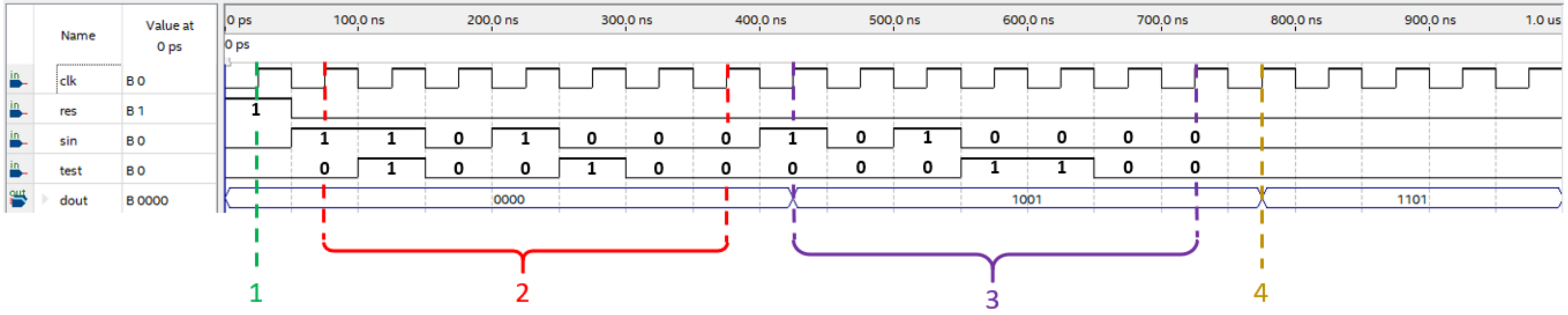
5.3. ModelSim Waveform



1. Reset registers to known state
2. Serially load in “0001011” and introduce single bit error
3. Serially load in “0000101” and introduce single bit error

   dout = 1101 which matches the data bits (first 4 bits) of the 7-bit message load in during step 2 hence the single bit error was corrected successfully
4. dout = 0101 which matches the data bits of the 7-bit message load in during step 3 hence the single bit error was corrected successfully

To prove the test signal is defiantly introducing an error, the next waveform shows what happens when a multiple bit error is introduced with the same input data



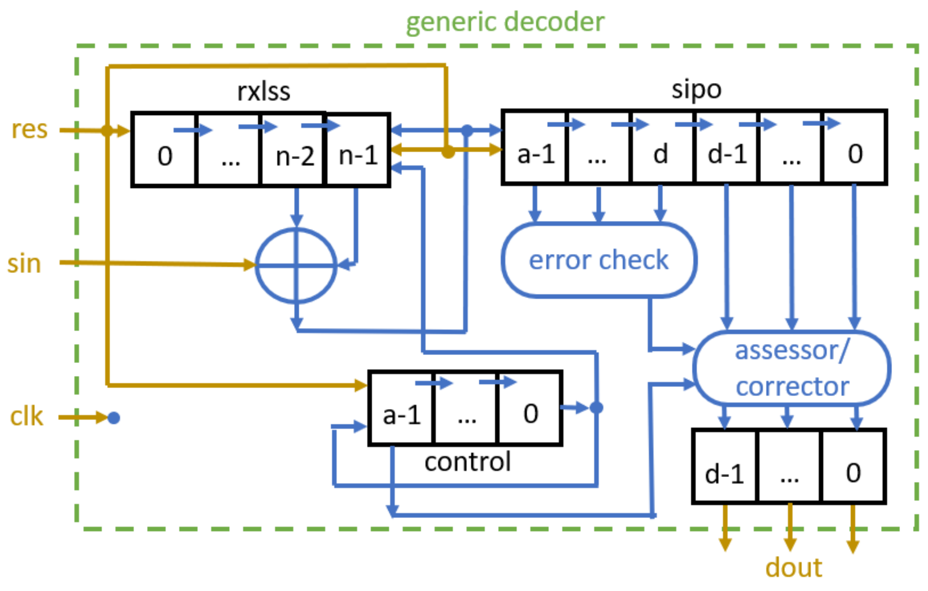
1. Reset registers to known state
2. Serially load in “0001011” and introduce single bit error
3. Serially load in “0000101” and introduce single bit error

   dout = 1001 which doesn’t match the data bits of the 7-bit message load in during step 2 hence the test pin is introducing errors correctly
4. dout = 1101 which doesn’t match the data bits of the 7-bit message load in during step 3 hence test pin is introducing errors correctly

5. Generic Decoder

6.1. Design

Key			
🟩	Entity	n	Error Bits
🟧	Port	d	Data Bits ($2^{n-(n+1)}$)
🟦	Sequential Logic (dependent on clk)	a	All Bits ($n + 2^{n-(n+1)}$)
⬛	Register		



The diagram above illustrates the operation of the generic decoder which should be the same as that of the decoders described in **sections 1** and **2**. The difference being that the VHDL code for the generic decoder can be changed from a 7/4 to a 15/11 decoder by changing a single generic value. In this case, the generic value that is altered is the number of error bits used. For a 7/4 decoder this would be 3 and for a 15/11 it would be 4. Error bits was chosen over other generic values as it ensures the coding system is perfect.

The Boolean expression for the logic which decodes the input signal is:

$$sipo(a-1)\,\&\,rxlss(0) = sin \oplus rxlss(n-2) \oplus rxlss(n-1)$$

$$\boldsymbol{Where}:$$

$$n\ =\ number\ of\ error\ bits$$

$$a\ =\ number\ of\ bits\ in\ a\ complete\ message\ (n + 2^{n-(n+1)})$$

When the number of error bits is set to 4, the impulse response and LUT is identically to that of the 15/11 decoder described in **section 2** on **page 9**. However, when the number of error bits is set to 3 the impulse response and LUT differs from that of the 7/4 decoder describe in **section 1** on **page 3** as the generic decoder always XOR's the two MSB's of rxlss rather than the MSB and LSB. This was done to keep the VHDL code as generic as possible. The only part of the VHDL code that isn't completely generic is the error check and assessor/corrector process as it was out of scope for this assignment.

This impulse response for the generic decoder when the number of error bits is set to 3 is:

sin	$rxlss(0)$	$rxlss(1)$	$rxlss(2)$	$sipo(6)\ \&\ rxlss(0)$
1	0	0	0	1
0	1	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
0	0	1	1	0

From this impulse response, the Noise Look-Up table (**LUT**) can be given as:

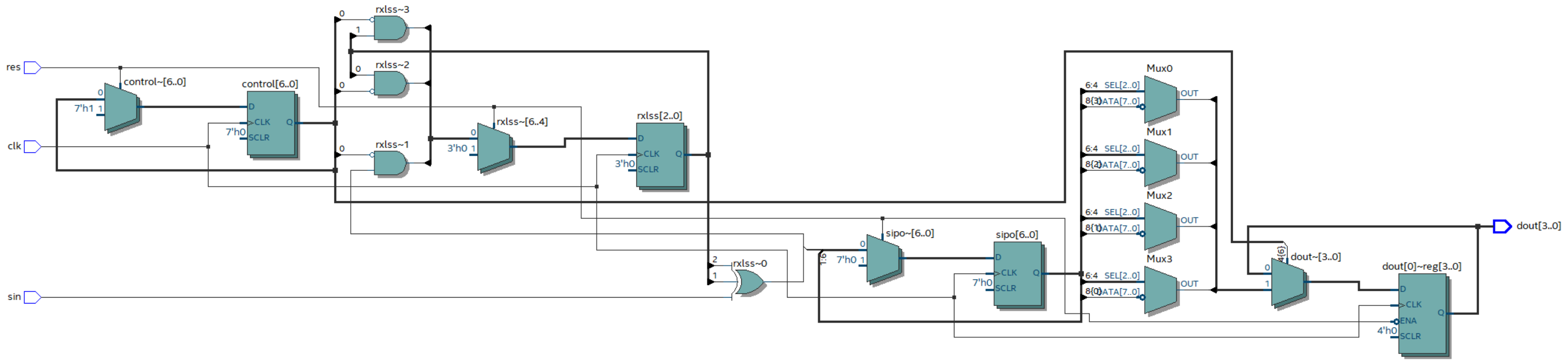
sin	$sipo$
	6 5 4 3 2 1 0
0000001	0 0 1 1 1 0 1
0000010	0 1 1 1 0 1 0
0000100	1 1 1 0 1 0 0
0001000	1 1 0 1 0 0 0
0010000	1 0 1 0 0 0 0
0100000	0 1 0 0 0 0 0
1000000	1 0 0 0 0 0 0

- error code
- correction code
- ignored (data = "0000")

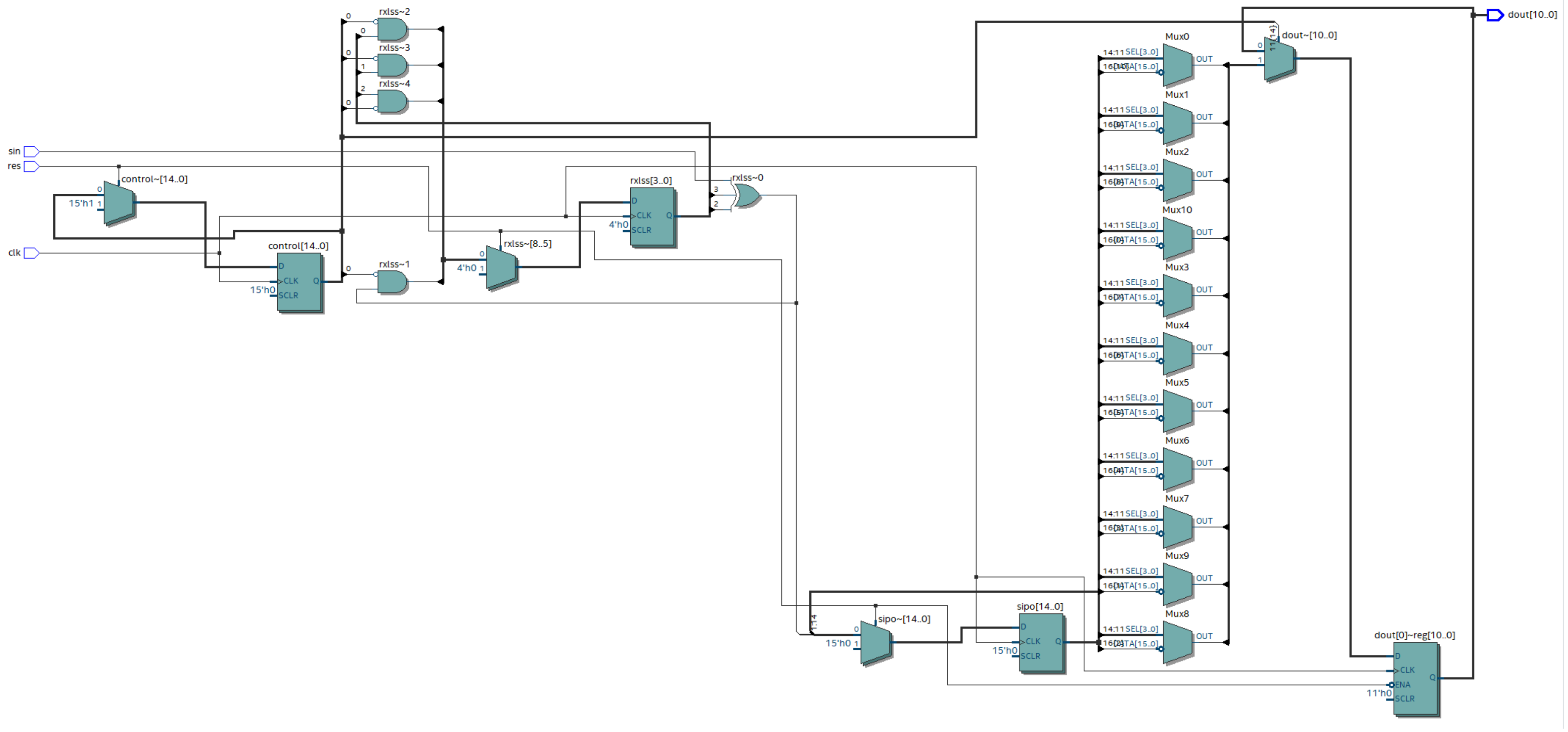
6.2. VHDL Code

```vhdl
library IEEE; -- include ieee library
use IEEE.STD_LOGIC_1164.all; -- include STD_LOGIC_1164 package from IEEE library to use the std_logic and
                             -- std_logic_vector data types which adds U (undefined) and Z (high impedance) assignments to the
                             -- standard VHDL bit and bit_vector data types
use IEEE.STD_LOGIC_UNSIGNED.all; -- Include STD_LOGIC_UNSIGNED package from IEEE library to be able to perform arithmetic,
                                 -- conversion and comparison operations on the std_logic_vector data type

-- define the interface between the generic decoder and its external environment
ENTITY decoderGeneric IS
   GENERIC (errorBits   : NATURAL := 3); -- define the number of error bits e.g. if set to 3 then the decoder will be a 7/4
                                         -- decoder
   PORT (
      clk, res, sin  : IN STD_LOGIC; -- define the single point input ports (clock, reset & serial-in)
      dout           : OUT STD_LOGIC_VECTOR(2**errorBits - (errorBits + 2) downto 0) -- define the output port (data-out) and
                                                                                     -- set it to be as wide as the number of
                                                                                     -- data bits
   );
END decoderGeneric;

-- define the internal organisation and operation of the generic decoder
ARCHITECTURE rtl OF decoderGeneric IS
   -- architecture declarations

   -- calculate the number of data bits and total number of bits (error bits + data bits) given the number of error bits and
   -- define them as constants
   CONSTANT dataBits : NATURAL                                    := 2**errorBits - (errorBits + 1);
   CONSTANT allBits  : NATURAL                                    := errorBits + dataBits;

   SIGNAL   rxlss    : STD_LOGIC_VECTOR(errorBits - 1 downto 0)   := (errorBits - 1 downto 0 => '0'); -- define the rxlss
                                                                                                    -- (receive-linear-
                                                                                                    -- sequential-system)
                                                                                                    -- shift register and
                                                                                                    -- set it to be as wide
                                                                                                    -- as the number of
                                                                                                    -- error bits
   SIGNAL   control  : STD_LOGIC_VECTOR(allBits - 1 downto 0); -- define the control shift register and set it to be as
                                                               -- wide as the total number of bits
   SIGNAL   sipo     : STD_LOGIC_VECTOR(allBits - 1 downto 0)     := (allBits - 1 downto 0 => '0'); -- define the sipo
                                                                                                  -- (serial-in-parallel-
                                                                                                  -- out) shift register
                                                                                                  -- and set set it to be
                                                                                                  -- as wide as the total
                                                                                                  -- number of bits
   -- concurrent statements
   BEGIN
      -- define all linear sequential logic as a single process
      PROCESS BEGIN
         WAIT UNTIL RISING_EDGE (clk); -- ensures that each line of code in this process is dependent on a rising clock edge

         -- when the reset port is set high, reset all registers to known state
         IF (res = '1') THEN
            control <= '1' & (allBits - 2 downto 0 => '0');  -- set the MSB high and all other bits low
            rxlss <= (errorBits - 1 downto 0 => '0'); -- set all bits low
            sipo <= (allBits - 1 downto 0 => '0');  -- set all bits low
         -- when the reset port is set low, decode data coming in on the sin port sequentially and correct single bit errors
         ELSIF (res = '0') THEN
            -- decode the data coming in on the sin port by assigning the LSB of the rxlss register to the sin port XOR'ed
            -- with the two MSB's of the rxlss regiester. However, when all bits have been received on the sin port (MSB of
            -- the control register is low) set the LSB of the rxlss register low
            rxlss(0) <= (sin XOR rxlss(errorBits - 2) XOR rxlss(errorBits - 1)) AND NOT(control(0));

            -- shift the current value of each bit in the rxlss register up to the next MSB but set them low when all bits
            -- have been received
            FOR i IN 1 TO errorBits - 1 LOOP
               rxlss(i) <= rxlss(i - 1) AND NOT(control(0));
            END LOOP;

            -- store decoded data in sipo register by assigning the MSB to the sin port XOR'ed with the two MSB's of the
            -- rxlss register and shifting the current value of each bit down to the next LSB
            sipo <= (sin XOR rxlss(errorBits - 2) XOR rxlss(errorBits - 1)) & sipo(allBits - 1 downto 1);

            -- keep track of the number of bits received sequentially on the sin port by shifting each bit of the control
            -- register to the next LSB and looping the LSB back round to the MSB (a single bit will always be high while the
            -- others are low)
            control <= control(0) & control(allBits - 1 downto 1);

            -- when the number of the error bits has been set to 3 and the sipo register has been filled (all 7-bits have been
            -- received on sin port and decoded), comapre the 3 MSB's of the sipo register (error bits) with the noise look-up
            -- table for a 7/4 decoder
            IF (control(allBits - 1) = '1') AND errorBits = 3 THEN
               CASE (sipo(allBits - 1 downto dataBits)) IS
                  -- when single bit error is present XOR data bits of sipo with appropriate value from look-up table and send
                  -- out the corrected data in parallel on the dout port
                  WHEN "001" => dout <= (sipo(3 downto 0)) XOR ("1101");
                  WHEN "011" => dout <= (sipo(3 downto 0)) XOR ("1010");
                  WHEN "111" => dout <= (sipo(3 downto 0)) XOR ("0100");
                  WHEN "110" => dout <= (sipo(3 downto 0)) XOR ("1000");
                  -- when no single bit error is present or the data bits of sipo are "000" send out the data bits of sipo in
                  -- parallel on the dout port
                  WHEN OTHERS => dout <= sipo(3 downto 0);
               END CASE;
            -- when the number of the error bits has been set to 4 and the sipo register has been filled (all 15-bits have been
            -- received on sin port and decoded), comapre the 4 MSB's of the sipo register (error bits) with the noise look-up
            -- table for a 15/11 decoder
            ELSIF (control(allBits - 1) = '1') AND errorBits = 4 THEN
               CASE (sipo(allBits - 1 downto dataBits)) IS
                  -- when single bit error is present XOR data bits of sipo with appropriate value from look-up table and send
                  -- out the corrected data in parallel on the dout port
                  WHEN "0001" => dout <= (sipo(10 downto 0)) XOR ("11101011001");
                  WHEN "0011" => dout <= (sipo(10 downto 0)) XOR ("11010110010");
                  WHEN "0111" => dout <= (sipo(10 downto 0)) XOR ("10101100100");
                  WHEN "1111" => dout <= (sipo(10 downto 0)) XOR ("01011001000");
                  WHEN "1110" => dout <= (sipo(10 downto 0)) XOR ("10110010000");
                  WHEN "1101" => dout <= (sipo(10 downto 0)) XOR ("01100100000");
                  WHEN "1010" => dout <= (sipo(10 downto 0)) XOR ("11001000000");
                  WHEN "0101" => dout <= (sipo(10 downto 0)) XOR ("10010000000");
                  WHEN "1011" => dout <= (sipo(10 downto 0)) XOR ("00100000000");
                  WHEN "0110" => dout <= (sipo(10 downto 0)) XOR ("01000000000");
                  WHEN "1100" => dout <= (sipo(10 downto 0)) XOR ("10000000000");
                  -- when no single bit error is present or the data bits of sipo are "000" send out the data bits of sipo in
                  -- parallel on the dout port
                  WHEN OTHERS => dout <= sipo(10 downto 0);
               END CASE;
            END IF;
         END IF;
      END PROCESS;
END rtl;
```

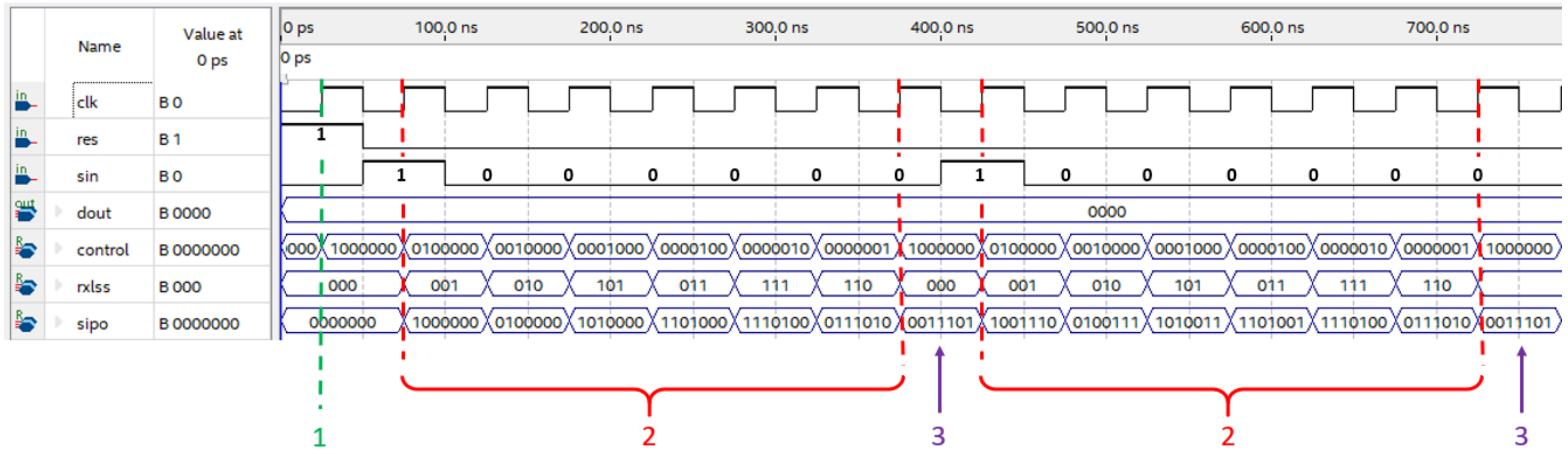
Compiled to schematic when errorBits = 3   (7/4 decoder)



Compiled to schematic when errorBits = 4   (15/11 decoder)



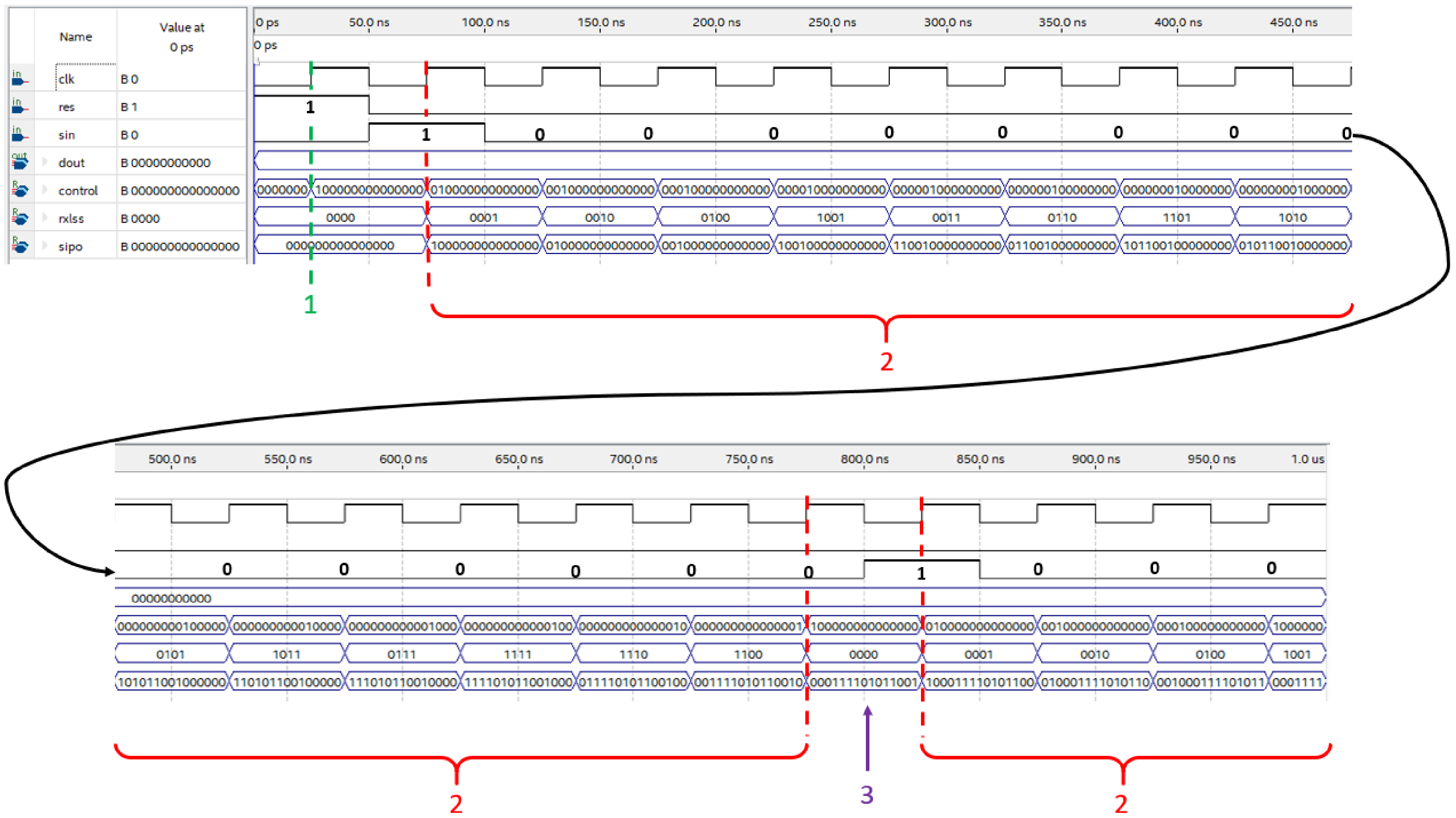
6.3. ModelSim Waveform



1. Reset registers to known state
2. Serially load in "0000001"
3. After 7-bits have been decoded (control = "1000000"), sipo = "0011101" which matches the impulse response on page 29

dout = "0000" throughout as sipo contains the error code "001" from the LUT which means the data bits from sipo (four LSB's) are XOR'ed with the corresponding correction code from the LUT

$$\overset{sipo}{1101} \oplus \overset{code}{1101} = \overset{dout}{0000}$$



3. Reset registers to known state
4. Serially load in "000000000000001"

4. After 15-bits have been decoded (control = “100000000000000”), sipo = “000111101011001” which matches the impulse response on page 12

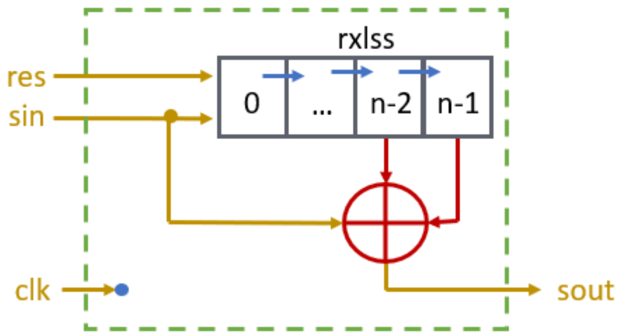
   dout = “00000000000” throughout as sipo contains the error code “0001” from the LUT which means the data bits from sipo (eleven LSB’s) are XOR’ed with the corresponding correction code from the LUT

$$\underset{sipo}{11101011001} \oplus \underset{code}{11101011001} = \underset{dout}{00000000000}$$

6. Generic Encoder

7.1. Design

Key			
🟩	Entity	n	Error Bits
🟧	Port		
🟦	Sequential Logic (dependent on clk)		
⬛	Register		
🟥	Combination Logic		



The diagram above illustrates the operation of the generic encoder which should be the same as that of the encoder described in **sections 3**. The difference being that the VHDL code for the generic encoder can be changed from a 7/4 to a 15/11 encoder by changing a single generic value. Again, the generic value that is altered is the number of error bits used. For a 7/4 encoder this would be 3 and for a 15/11 it would be 4. Error bits was chosen over other generic values as it ensures the coding system is perfect.

The Boolean expression for the logic which encodes the input signal is:

$$sout = sin \oplus rxlss(n-2) \oplus rxlss(n-1)$$

$$\boldsymbol{Where}:$$

$$n = number\ of\ error\ bits$$

When the number of error bits is set to 3 the impulse response and LUT differs from that of the 7/4 encoder describe in **section 3** on **page 18** as the generic encoder always XOR's the two MSB's of rxlss rather than the MSB and LSB. This was done to keep the VHDL code for the encoder completely generic.

This impulse response for the generic encoder when the number of error bits is set to 3 is:

sin	$rxlss(0)$	$rxlss(1)$	$rxlss(2)$	$sout$
1	0	0	0	1
0	1	0	0	0
0	0	1	0	1
0	0	0	1	1
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

From this impulse response, the Noise Look-Up table (**LUT**) can be given as:

sin	$sout$
0000001	0001101
0000010	0011010
0000100	0110100
0001000	1101000
0010000	1010000
0100000	0100000
1000000	1000000

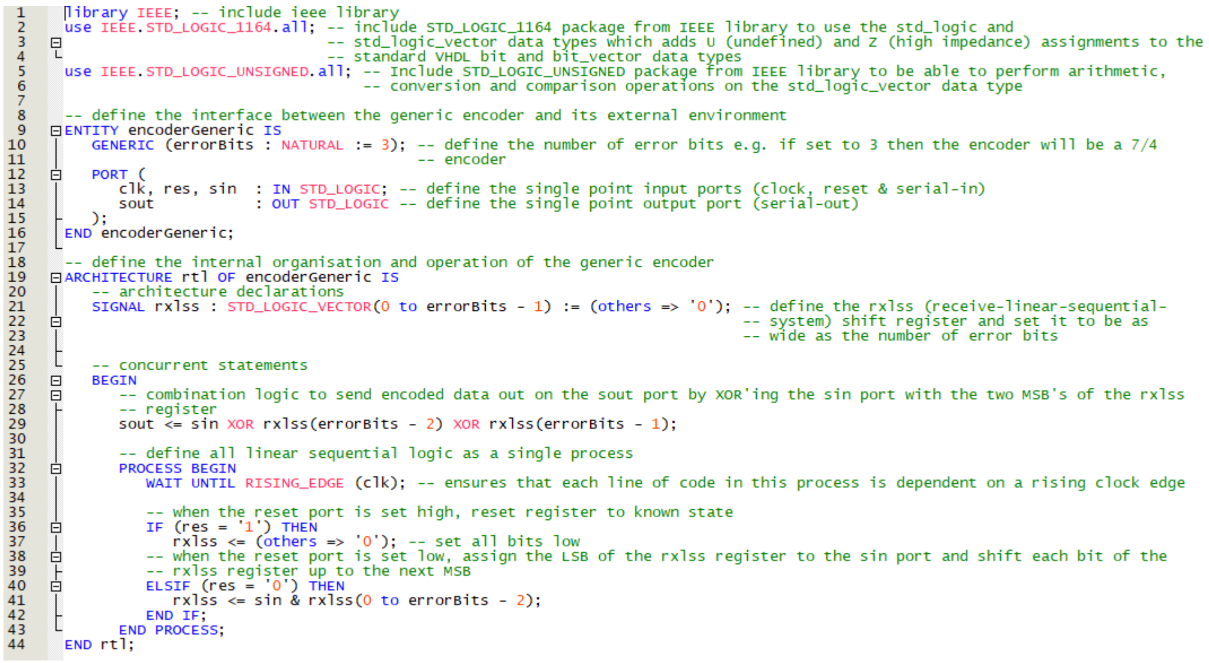
The impulse response for the generic encoder when the number of error bits is set to 4 is:

sin	$rxlss(0)$	$rxlss(1)$	$rxlss(2)$	$rxlss(3)$	$sout$
1	0	0	0	0	1
0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	1
0	0	0	0	1	1
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

From this impulse response, the Noise Look-Up table (**LUT**) can be given as:

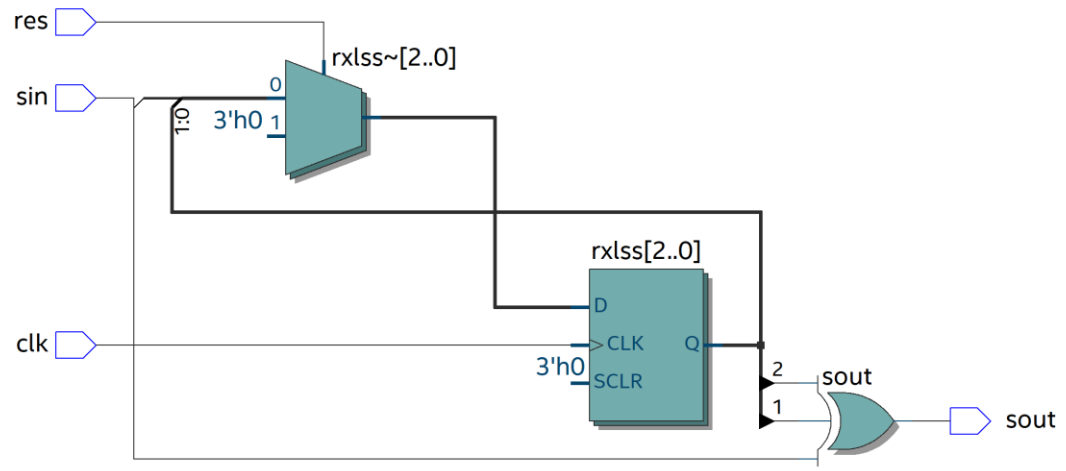
sin	*sout*
000000000000001	000000000011001
000000000000010	000000000110010
000000000000100	000000001100100
000000000001000	000000011001000
000000000010000	000000110010000
000000000100000	000001100100000
000000001000000	000011001000000
000000010000000	000110010000000
000000100000000	001100100000000
000001000000000	011001000000000
000010000000000	110010000000000
000100000000000	100100000000000
001000000000000	001000000000000
010000000000000	010000000000000
100000000000000	100000000000000

7.2. VHDL Code

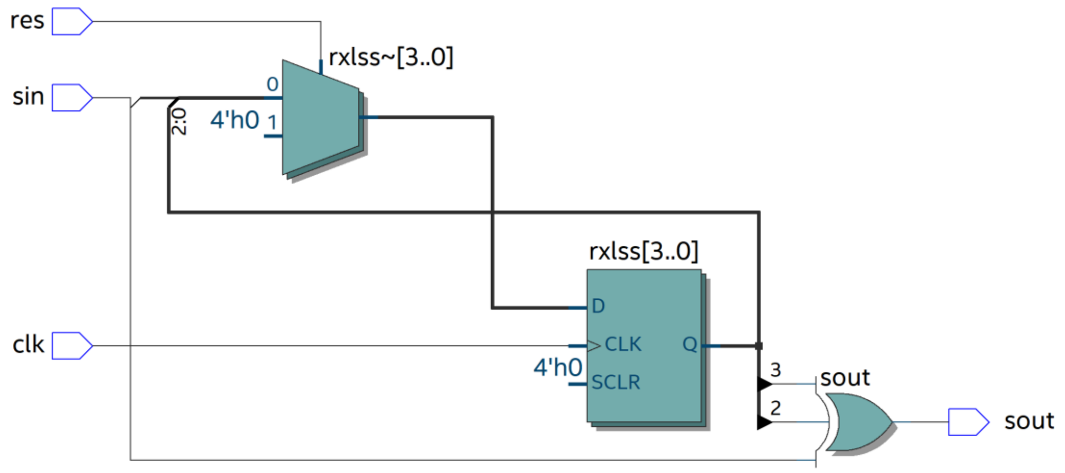


```vhdl
library IEEE; -- include ieee library
use IEEE.STD_LOGIC_1164.all; -- include STD_LOGIC_1164 package from IEEE library to use the std_logic and
                             -- std_logic_vector data types which adds U (undefined) and Z (high impedance) assignments to the
                             -- standard VHDL bit and bit_vector data types
use IEEE.STD_LOGIC_UNSIGNED.all; -- Include STD_LOGIC_UNSIGNED package from IEEE library to be able to perform arithmetic,
                                 -- conversion and comparison operations on the std_logic_vector data type

-- define the interface between the generic encoder and its external environment
ENTITY encoderGeneric IS
   GENERIC (errorBits : NATURAL := 3); -- define the number of error bits e.g. if set to 3 then the encoder will be a 7/4
                                       -- encoder
   PORT (
      clk, res, sin  : IN STD_LOGIC; -- define the single point input ports (clock, reset & serial-in)
      sout           : OUT STD_LOGIC -- define the single point output port (serial-out)
   );
END encoderGeneric;

-- define the internal organisation and operation of the generic encoder
ARCHITECTURE rtl OF encoderGeneric IS
   -- architecture declarations
   SIGNAL rxlss : STD_LOGIC_VECTOR(0 to errorBits - 1) := (others => '0'); -- define the rxlss (receive-linear-sequential-
                                                                            -- system) shift register and set it to be as
                                                                            -- wide as the number of error bits


   -- concurrent statements
   BEGIN
      -- combination logic to send encoded data out on the sout port by XOR'ing the sin port with the two MSB's of the rxlss
      -- register
      sout <= sin XOR rxlss(errorBits - 2) XOR rxlss(errorBits - 1);

      -- define all linear sequential logic as a single process
      PROCESS BEGIN
         WAIT UNTIL RISING_EDGE (clk); -- ensures that each line of code in this process is dependent on a rising clock edge

         -- when the reset port is set high, reset register to known state
         IF (res = '1') THEN
            rxlss <= (others => '0'); -- set all bits low
         -- when the reset port is set low, assign the LSB of the rxlss register to the sin port and shift each bit of the
         -- rxlss register up to the next MSB
         ELSIF (res = '0') THEN
            rxlss <= sin & rxlss(0 to errorBits - 2);
         END IF;
      END PROCESS;
END rtl;
```

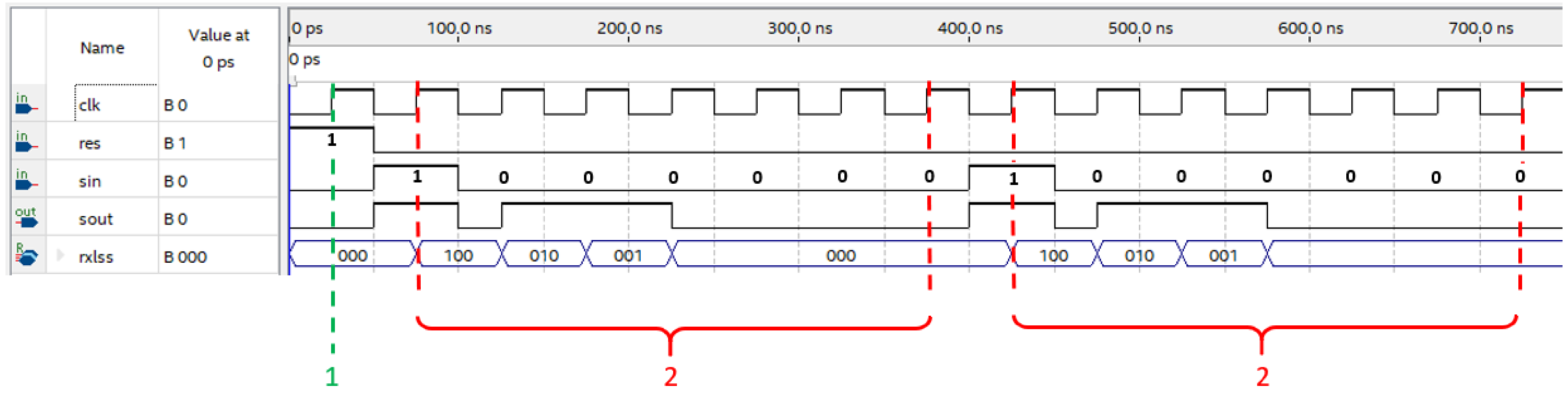
Compiled to schematic when errorBits = 3   (7/4 encoder)



Compiled to schematic when errorBits = 4   (15/11 encoder)

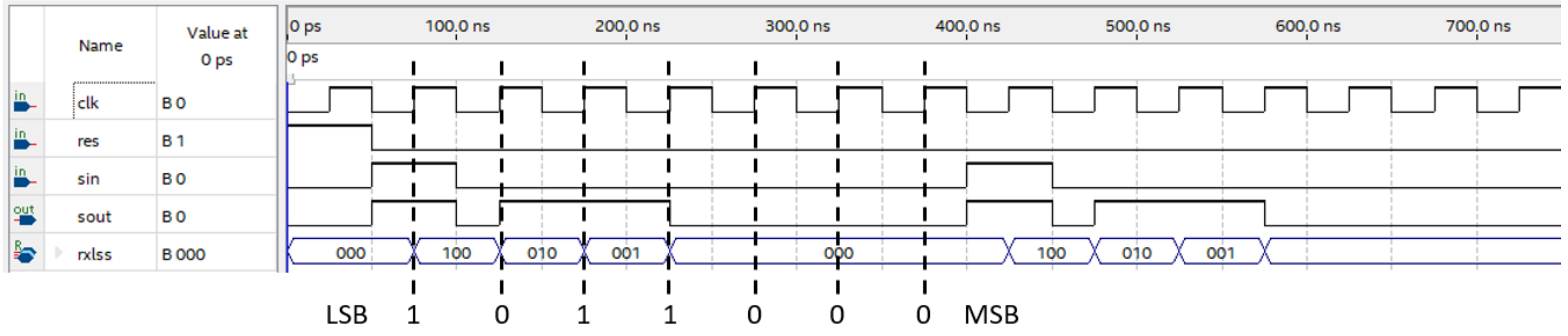


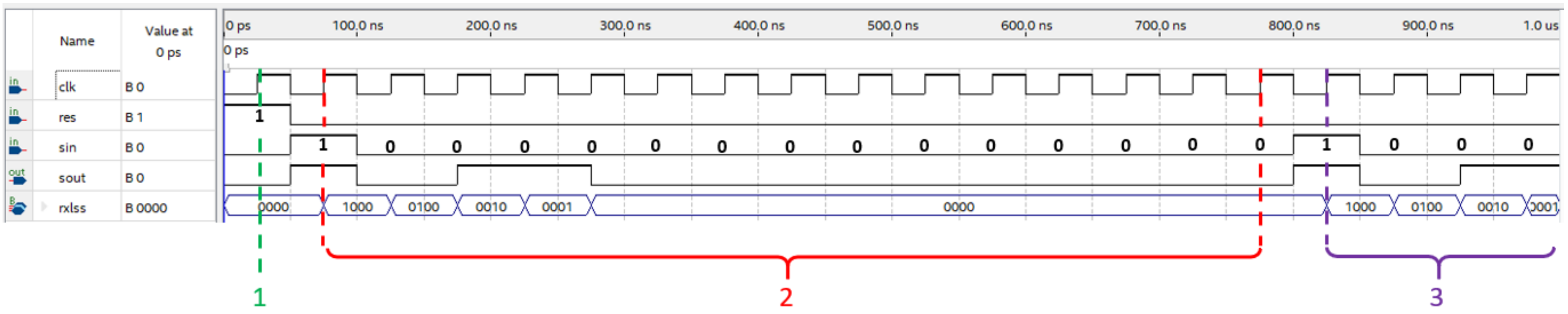
7.3. ModelSim Waveform



1. Reset registers to known state
2. Serially load in “0000001” and send encoded signal out

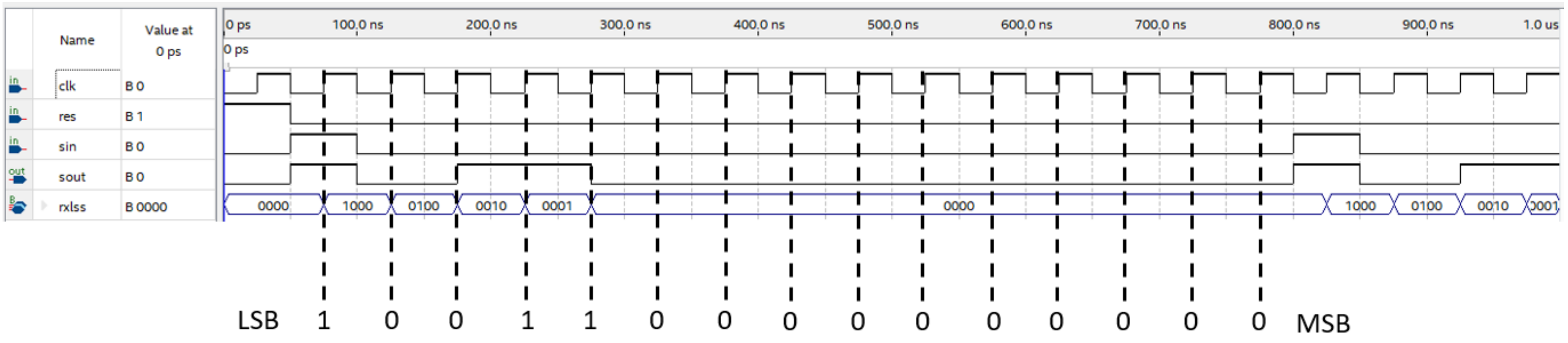
Due to the delay in the combination logic, the decoder will read falling edges on sout as “1” and rising edges as “0” hence sout matches the impulse response “0001101” seen on **page 17**





1. Reset registers to known state
2. Serially load in "000000000000001" and send encoded signal out

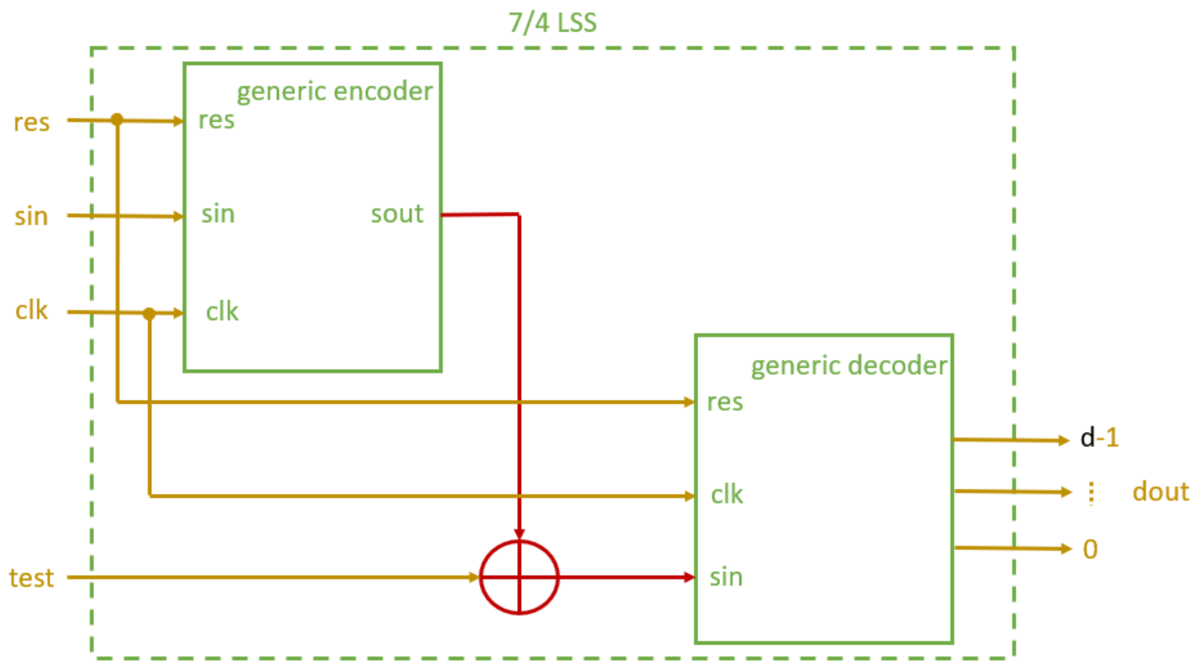
Due to the delay in the combination logic, the decoder will read falling edges on sout as "1" and rising edges as "0" hence sout matches the impulse response "000000000011001" seen on **page 17**



7. Generic Linear Sequential Coding System

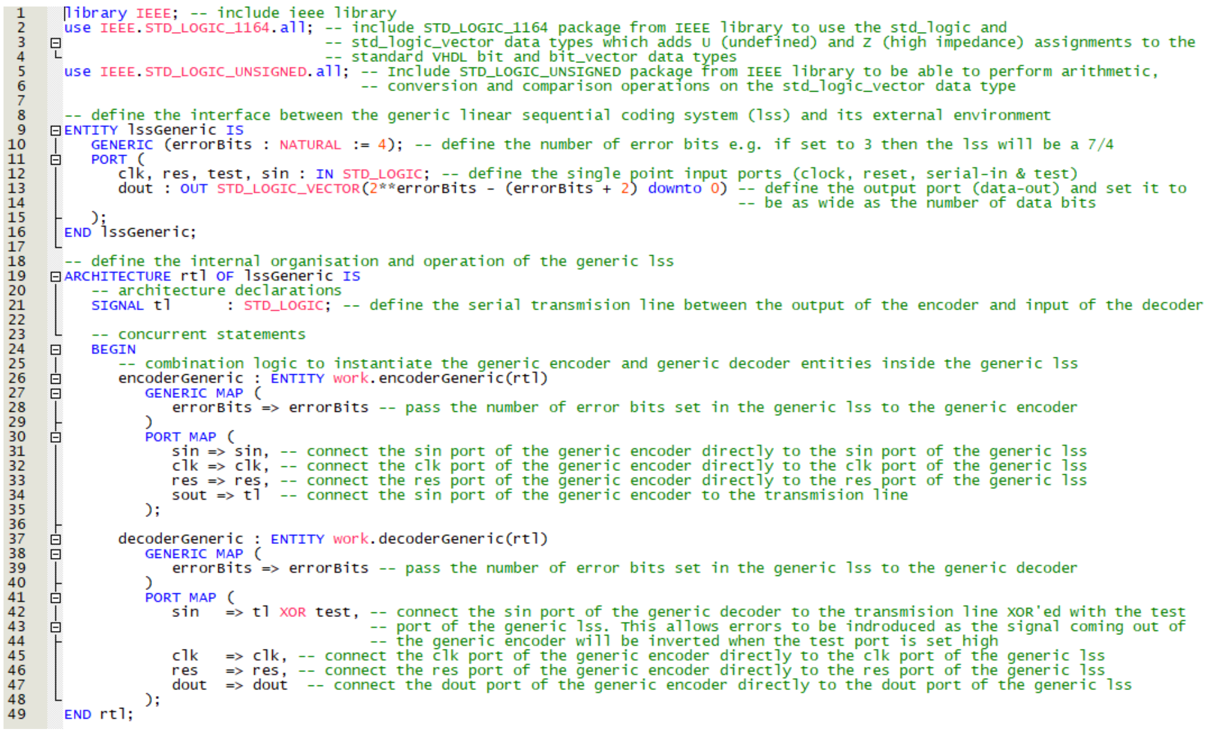
8.1. Design

Key			
🟩	Entity	n	Error Bits
🟧	Port	d	Data Bits ($2^{n-(n+1)}$)
🟦	Sequential Logic (dependent on clk)		
⬛	Register		
🟥	Combination Logic		



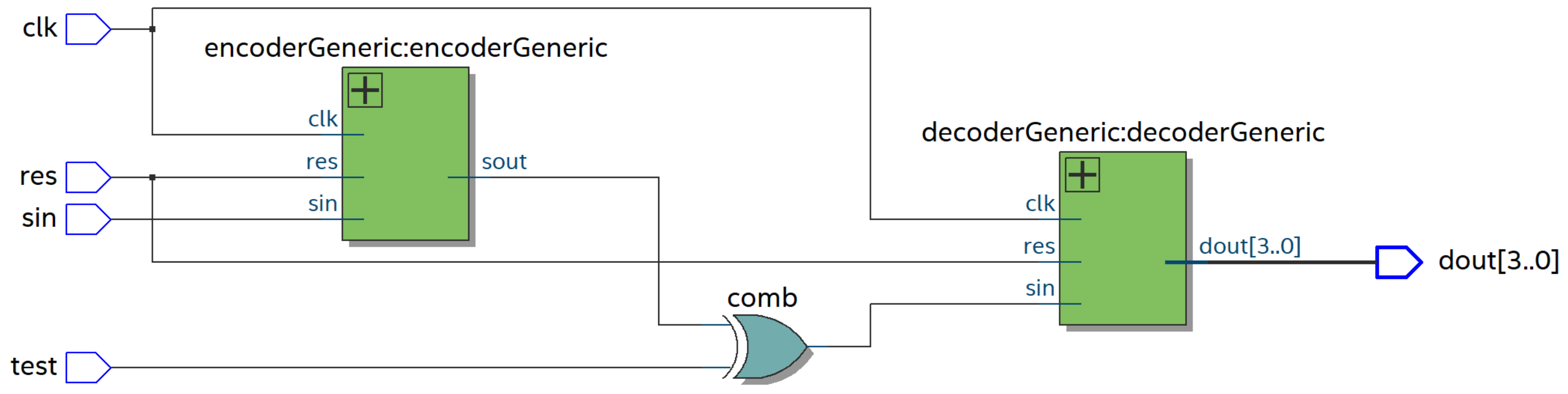
The diagram above illustrates the operation of the generic LSS which should be the same as that of the LSS described in **sections 4**. The difference being that the VHDL code for the generic encoder can be changed from a 7/4 to a 15/11 LSS by changing a single generic value. Again, the generic value that is altered is the number of error bits used. For a 7/4 LSS this would be 3 and for a 15/11 it would be 4. Error bits was chosen over other generic values as it ensures the coding system is perfect.

8.2. VHDL Code

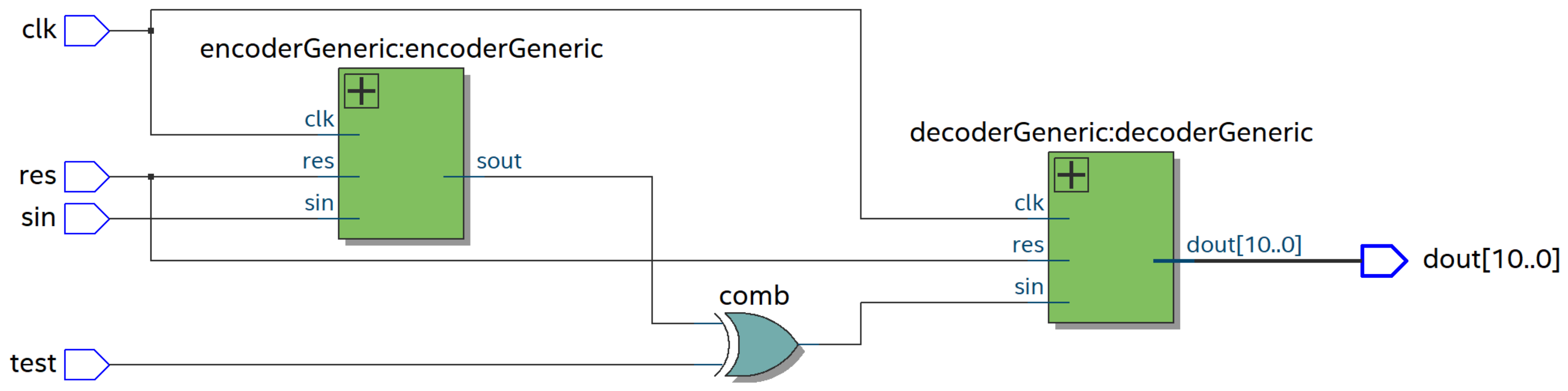


```vhdl
library IEEE; -- include ieee library
use IEEE.STD_LOGIC_1164.all; -- include STD_LOGIC_1164 package from IEEE library to use the std_logic and
                             -- std_logic_vector data types which adds U (undefined) and Z (high impedance) assignments to the
                             -- standard VHDL bit and bit_vector data types
use IEEE.STD_LOGIC_UNSIGNED.all; -- Include STD_LOGIC_UNSIGNED package from IEEE library to be able to perform arithmetic,
                                 -- conversion and comparison operations on the std_logic_vector data type

-- define the interface between the generic linear sequential coding system (lss) and its external environment
ENTITY lssGeneric IS
   GENERIC (errorBits : NATURAL := 4); -- define the number of error bits e.g. if set to 3 then the lss will be a 7/4
   PORT (
      clk, res, test, sin : IN STD_LOGIC; -- define the single point input ports (clock, reset, serial-in & test)
      dout : OUT STD_LOGIC_VECTOR(2**errorBits - (errorBits + 2) downto 0) -- define the output port (data-out) and set it to
                                                                           -- be as wide as the number of data bits
   );
END lssGeneric;

-- define the internal organisation and operation of the generic lss
ARCHITECTURE rtl OF lssGeneric IS
   -- architecture declarations
   SIGNAL tl      : STD_LOGIC; -- define the serial transmision line between the output of the encoder and input of the decoder

   -- concurrent statements
   BEGIN
      -- combination logic to instantiate the generic encoder and generic decoder entities inside the generic lss
      encoderGeneric : ENTITY work.encoderGeneric(rtl)
         GENERIC MAP (
            errorBits => errorBits -- pass the number of error bits set in the generic lss to the generic encoder
         )
         PORT MAP (
            sin => sin, -- connect the sin port of the generic encoder directly to the sin port of the generic lss
            clk => clk, -- connect the clk port of the generic encoder directly to the clk port of the generic lss
            res => res, -- connect the res port of the generic encoder directly to the res port of the generic lss
            sout => tl  -- connect the sin port of the generic encoder to the transmision line
         );

      decoderGeneric : ENTITY work.decoderGeneric(rtl)
         GENERIC MAP (
            errorBits => errorBits -- pass the number of error bits set in the generic lss to the generic decoder
         )
         PORT MAP (
            sin   => tl XOR test, -- connect the sin port of the generic decoder to the transmision line XOR'ed with the test
                                  -- port of the generic lss. This allows errors to be indroduced as the signal coming out of
                                  -- the generic encoder will be inverted when the test port is set high
            clk   => clk, -- connect the clk port of the generic encoder directly to the clk port of the generic lss
            res   => res, -- connect the res port of the generic encoder directly to the res port of the generic lss
            dout  => dout  -- connect the dout port of the generic encoder directly to the dout port of the generic lss
         );
END rtl;
```

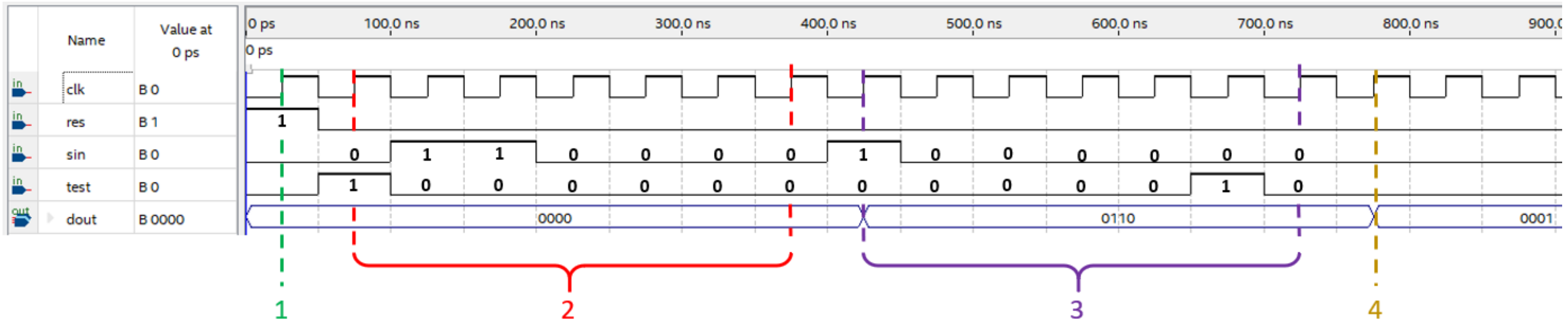
Compiled to schematic when errorBits = 3   (7/4 LSS)



Compiled to schematic when errorBits = 4   (15/11 LSS)

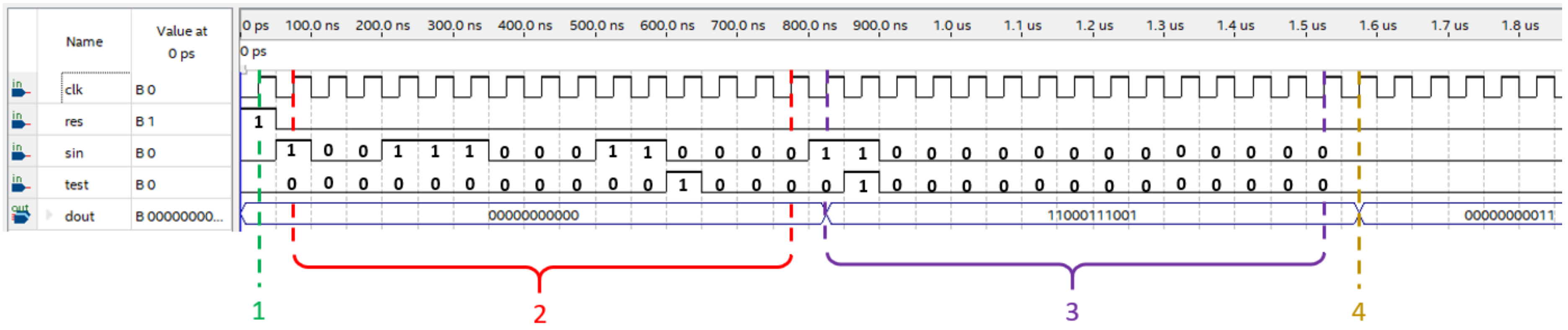


8.3. ModelSim Waveform



1. Reset registers to known state
2. Serially load in “0000110” and introduce single bit error
3. Serially load in “0000001” and introduce single bit error

   dout = 0110 which matches the data bits (first 4 bits) of the 7-bit message load in during step 2 hence the single bit error was corrected successfully
4. dout = 0001 which matches the data bits of the 7-bit message load in during step 3 hence the single bit error was corrected successfully



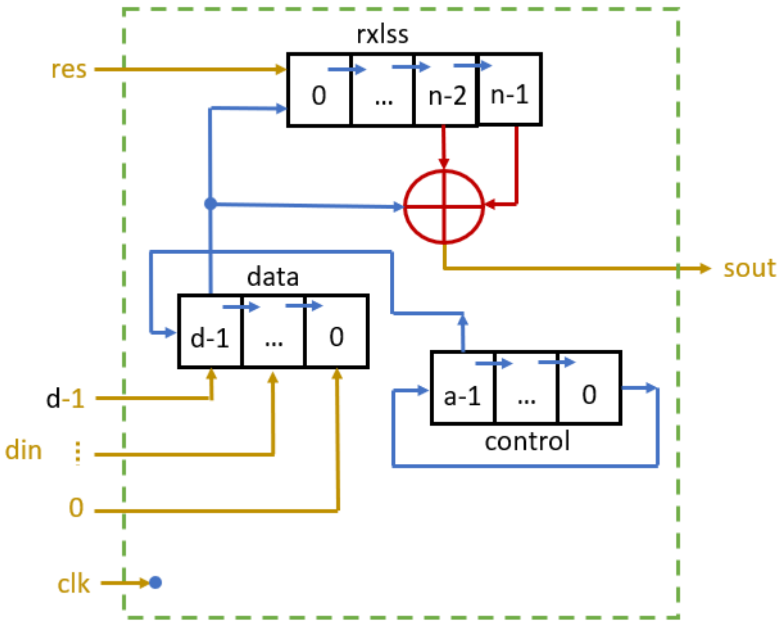
1. Reset registers to known state
2. Serially load in “000011000111001” and introduce single bit error
3. Serially load in “110000000000000” and introduce single bit error

   dout = 11000111001 which matches the data bits (first 11 bits) of the 15-bit message load in during step 2 hence the single bit error was corrected successfully
4. dout = 00000000001 which matches the data bits of the 15-bit message load in during step 3 hence the single bit error was corrected successfully

8. Parallel-In Encoder

9.1. Design

Key			
🟩	Entity	n	Error Bits
🟨	Port	d	Data Bits ($2^{n-(n+1)}$)
🟦	Sequential Logic (dependent on clk)	a	All Bits ($n + 2^{n-(n+1)}$)
⬛	Register		
🟥	Combination Logic		



One problem with the encoder used in all the previous sections is that the entire 7-bit message needs to be loaded in serially. This means the three errors bits (last three bits) could be set to “1” which would prevent the message from being correctly decoded. A solution to this problem is to load the data in parallel. The diagram above illustrates the operation of the parallel 7/4 encoder which should be:

1. Can be reset at any time by setting the res port high
2. Decodes messages sent in, in parallel on the din port
3. Outputs the encoded messages serially
4. Continues to operate once the message has been encoded i.e. once the first message has been encoded, any new data on din will be read automatically

The purpose of each component illustrated in the diagram above is:

- **din input port** – receives input message
- **clk input port** - receives clock signal
- **res input port** - receives reset signal
- **sout output port** – transmits encoded signal
- **$\oplus$ logic** – encodes input signal
- **rxlss shift register** – stores the last n bits input from the data register to encode the message

- **data shift register** – stores the input message
- **control shift register** – keeps track of the number of bits encoded. When MSB is "1" it loads signals on din into the data register

The Boolean expression for the logic which encodes the input signal is:

$$sout = data(d-1) \oplus rxlss(n-2) \oplus rxlss(n-1)$$

$$\boldsymbol{Where}:$$

$$n \ = \ number\ of\ error\ bits$$

$$d \ = \ number\ of\ data\ bits\ (2^{n-(n+1)})$$

The impulse response for the generic parallel encoder when the number of error bits is set to 3 is:

$data(3)$	$rxlss(0)$	$rxlss(1)$	$rxlss(2)$	$sout$
1	0	0	0	1
0	1	0	0	0
0	0	1	0	1
0	0	0	1	1
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

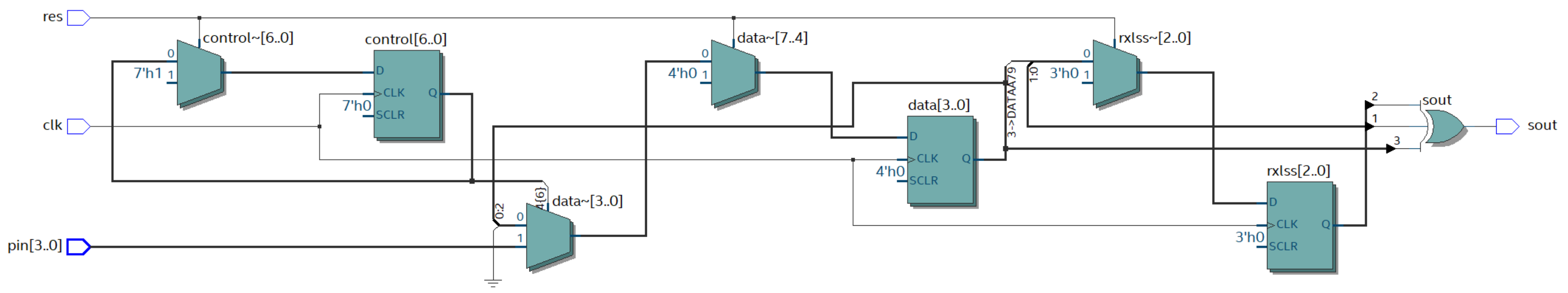
From this impulse response, the Noise Look-Up table (**LUT**) can be given as:

din	$sout$
0 1 2 3	
0 0 0 1	0001101
0 0 1 0	0011010
0 1 0 0	0110100
1 0 0 0	1101000

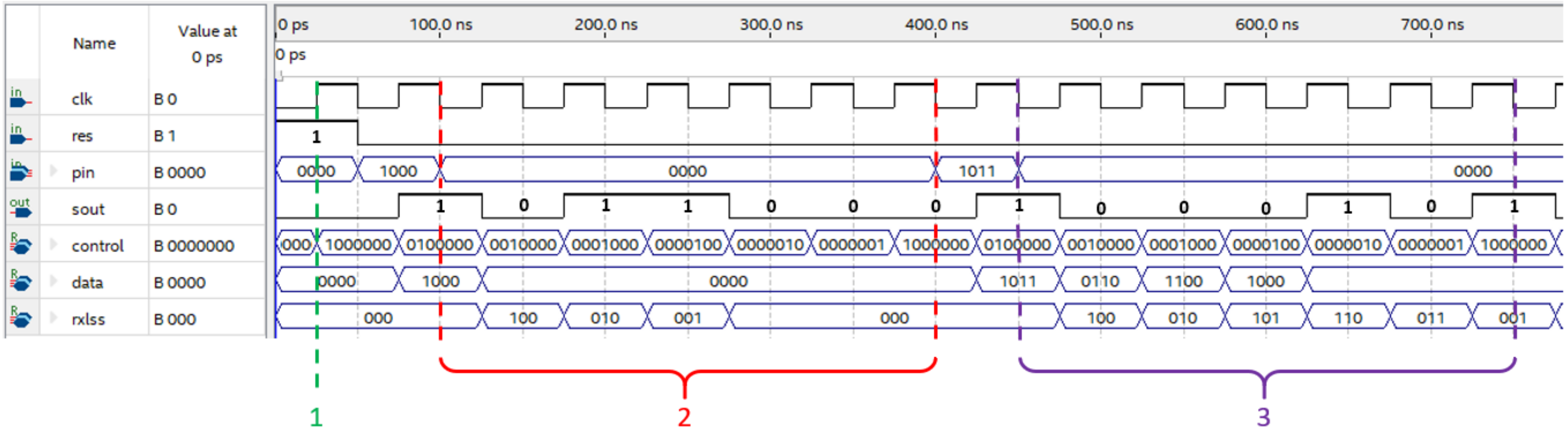
9.2. VHDL Code

```vhdl
library IEEE; -- include ieee library
use IEEE.STD_LOGIC_1164.all; -- include STD_LOGIC_1164 package from IEEE library to use the std_logic and
                             -- std_logic_vector data types which adds U (undefined) and Z (high impedance) assignments to the
                             -- standard VHDL bit and bit_vector data types
use IEEE.STD_LOGIC_UNSIGNED.all; -- Include STD_LOGIC_UNSIGNED package from IEEE library to be able to perform arithmetic,
                                 -- conversion and comparison operations on the std_logic_vector data type

-- define the interface between the generic encoder and its external environment
ENTITY pisoEncoderGeneric IS
   GENERIC (errorBits : NATURAL := 3); -- define the number of error bits e.g. if set to 3 then the encoder will be a 7/4
                                       -- encoder
   PORT (
      clk, res : IN STD_LOGIC; -- define the single point input ports (clock & reset)
      pin      : IN STD_LOGIC_VECTOR(2**errorBits - (errorBits + 2) downto 0); -- define the input port (parellel-in) and
                                                                               -- set it to be as wide as the number of
                                                                               -- data bits
      sout     : OUT STD_LOGIC -- define the single point output port (serial-out)
   );
END pisoEncoderGeneric;

-- define the internal organisation and operation of the generic encoder
ARCHITECTURE rtl OF pisoEncoderGeneric IS
   -- architecture declarations

   -- calculate the number of data bits and total number of bits (error bits + data bits) given the number of error bits and
   -- define them as constants
   CONSTANT dataBits : NATURAL                                 := 2**errorBits - (errorBits + 1);
   CONSTANT allBits  : NATURAL                                 := errorBits + dataBits;

   SIGNAL data       : STD_LOGIC_VECTOR(dataBits - 1 downto 0) := (others => '0'); -- define the data shift register and set
                                                                                   -- it to be as wide as the number of data
                                                                                   -- bits
   SIGNAL control    : STD_LOGIC_VECTOR(allBits - 1 downto 0); -- define the control shift register and set it to be as
                                                               -- wide as the total number of bits
   SIGNAL rxlss      : STD_LOGIC_VECTOR(0 to errorBits - 1)    := (others => '0'); -- define the rxlss (receive-linear-
                                                                                   -- sequential-system) shift register and
                                                                                   --  set it to be as wide as the number of
                                                                                   --  error bits

   -- concurrent statements
   BEGIN
      -- combination logic to send encoded data out on the sout port by XOR'ing the sin port with the two MSB's of the rxlss
      -- register
      sout <= data(dataBits - 1) XOR rxlss(errorBits - 2) XOR rxlss(errorBits - 1);

      -- define all linear sequential logic as a single process
      PROCESS BEGIN
         WAIT UNTIL RISING_EDGE (clk); -- ensures that each line of code in this process is dependent on a rising clock edge

         -- when the reset port is set high, reset register to known state
         IF (res = '1') THEN
            control <= '1' & (allBits - 2 downto 0 => '0');  -- set the MSB high and all other bits low
            rxlss <= (others => '0'); -- set all bits low
            data <= (others => '0'); -- set all bits low
         -- when the reset port is set low, shift data out of the data register serially and encode it
         ELSIF (res = '0') THEN
            -- assign the LSB of the rxlss register to the MSB of the data register and shift each bit of the rxlss register
            -- up to the next MSB
            rxlss(0) <= data(dataBits - 1);
            FOR i IN 1 TO errorBits - 1 LOOP
               rxlss(i) <= rxlss(i - 1);
            END LOOP;

            -- shift each bit of the data register up to the next MSB
            data <= data(dataBits - 2 downto 0) & '0';

            -- keep track of the number of bits shifted out of the data register sequentially by shifting each bit to the
            -- next LSB and looping the LSB back round to the MSB (a single bit will always be high while the others are low)
            control <= control(0) & control(allBits - 1 downto 1);

            -- when the encoder is reset or all bits of the previous data set have been encoded (MSB of the control register
            -- is high), load data on the pin port into the data register in parellel
            IF (control(allBits - 1) = '1') THEN
               FOR i IN 0 TO dataBits - 1 LOOP
                  data(i) <= pin(i);
               END LOOP;
            END IF;
         END IF;
      END PROCESS;
END rtl;
```

Compiled to schematic



9.3. ModelSim Waveform



1. Reset registers to known state
2. Load "0001" in parallel and send encoded signal out

sout matches that of the impulse response on **page 48** but unlike the other encoders, it is now bound by the clock as the input is loaded into a shift register that use sequential logic