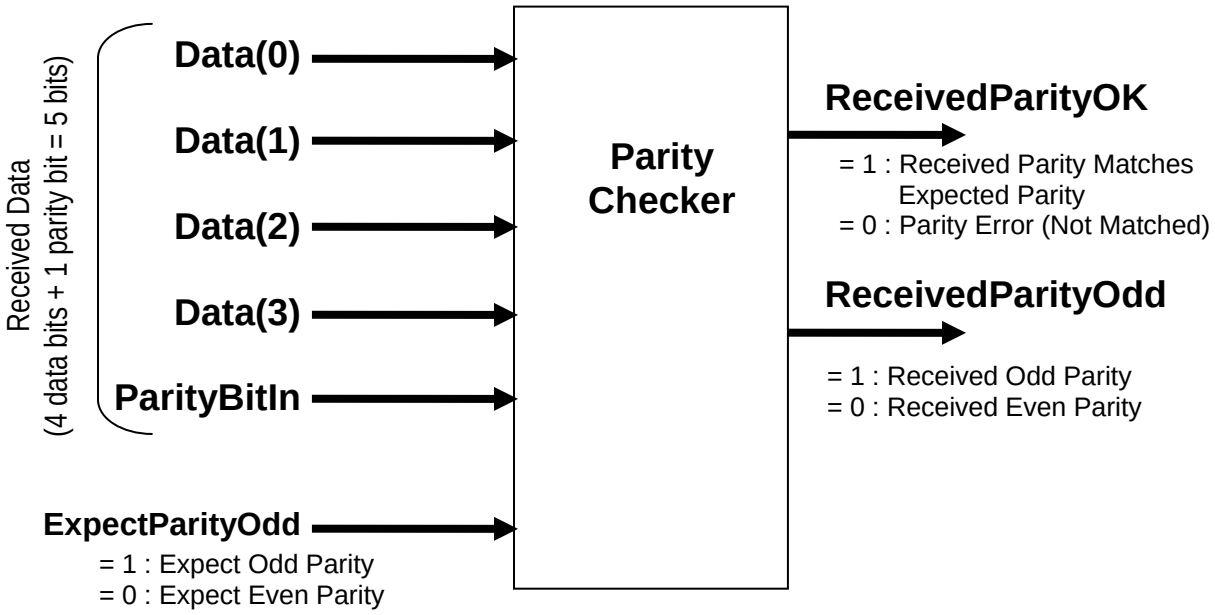
1. **Design a "Parity Checker" for a parallel data receiver <u>using VHDL</u> and a <u>Behavioral architecture</u>.**

The Parity Checker would be used to verify that data was received properly when it was transmitted from one location to another. This device takes in a 5-bit input, made up of 4 received data bits "Data(3..0)" and a received "Parity Bit In". It then computes and compares the parity of the <u>combined</u> 5-bit number it received with an "expected parity" for the data transmission. This expected parity is determined by another input signal (ExpectParityOdd).

The device generates an output signal that indicates the actual parity of the received 5-bits (ReceivedParityOdd), and a signal that indicates if the actual parity of the input matched the expected parity (ReceivedParityOK).

Use the diagram below to define the operation of your device:




```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ParityChecker is
   Port ( Data : in  STD_LOGIC_VECTOR (3 downto 0);
          ParityBit : in  STD_LOGIC;
          ExpectParityOdd : in  STD_LOGIC;
          ReceivedParityOK : out  STD_LOGIC;
          ReceivedParityOdd : out  STD_LOGIC);
end ParityChecker;
```

```
-- ONE POSSIBLE SOLUTION (OF MANY)
architecture Behavioral of ParityChecker is  begin

parity_check: process (Data, ParityBit, ExpectParityOdd) is
begin
        --Test Odd parity
       if ( (Data(0) XOR Data(1) XOR Data(2) XOR Data(3) XOR ParityBit) ='1')

       then --if Odd parity received
              ReceivedParityOdd <= '1';  -- Indicate Received Odd Parity
              if (ExpectParityOdd = '1')
                     then ReceivedParityOK <= '1';  -- Expected Odd and received Odd parity
                     else ReceivedParityOK <= '0';  -- Expected Even and received Odd parity
              end if;
       else --if Even parity received
              ReceivedParityOdd <= '0';  -- Indicate Received Even Parity
              if (ExpectParityOdd = '0')
                     then ReceivedParityOK <= '1'; --Expected Even and received Even parity
                     else ReceivedParityOK <= '0';  -- Expected Odd and received Even parity
              end if;
       end if;
end process parity_check;

end Behavioral;
```

```
-- ANOTHER POSSIBLE SOLUTION
architecture Behavioral of ParityChecker is   begin

parity_check: process (Data, ParityBit, ExpectParityOdd) is
begin
        ReceivedParityOK <= '0';  -- Indicate Error (Unless later see parity is OK).
                                  -- This will be overridden and have no effect if parity is OK
       --Test Odd parity
       if ( (Data(0) XOR Data(1) XOR Data(2) XOR Data(3) XOR ParityBit) ='1')

       then --if Odd parity received
              ReceivedParityOdd <= '1';  -- Indicate Received Odd Parity
              if (ExpectParityOdd = '1')
                     then ReceivedParityOK <= '1';  -- Expected Odd and received Odd parity
              end if;
       else --if Even parity received
              ReceivedParityOdd <= '0';  -- Indicate Received Even Parity
              if (ExpectParityOdd = '0')
                     then ReceivedParityOK <= '1'; --Expected Even and received Even parity
              end if;
       end if;
end process parity_check;
end Behavioral;
```

-- A COMPACT, BUT PROBLEMATIC SOLUTION

```
architecture Behavioral of ParityChecker is  begin

parity_check: process (Data, ParityBit, ExpectParityOdd) is
begin
        --Test Odd parity
        ReceivedParityOdd <= (Data(0) XOR Data(1) XOR Data(2) XOR Data(3) XOR ParityBit);

        if ( (ReceivedParityOdd = '1') AND (ExpectParityOdd = '1'))
                then ReceivedParityOK <= '1';  -- Expected Odd and received Odd parity

        elsif  ( (ReceivedParityOdd = '0') AND (ExpectParityOdd = '0') )
                then ReceivedParityOK <= '1';  -- Expected Even and received Even parity

        else ReceivedParityOK <= '0';  -- Otherwise, there was a parity error
        end if;
end process parity_check;

end Behavioral;
```

The VHDL code above actually suffers from at least 2 problems that arise from some of the unique (and sometimes frustrating!) features of VHDL that make it different from most other high-level programming languages.

1) First, we have the problem of updating a signal inside a PROCESS, and then expecting to use this <u>new</u> value in the logic for updating a second signal in the same process <u>before the 1st assignment has taken effect</u>. In this case,

ReceivedParityOdd <= (Data(0) XOR Data(1)…..

will not be executed until the entire Process is evaluated and the end of the process is reached. Therefore, the if ((**ReceivedParityOdd** = '1') ***and*** elsif ((**ReceivedParityOdd**='0') ***conditions are using a <u>previous</u> value for*** ReceivedParityOdd; **not** ***the value computed in the latest*** ReceivedParityOdd <= (Data(0) XOR Data(1)…..***statement (because this statement hasn't been executed yet!!)***

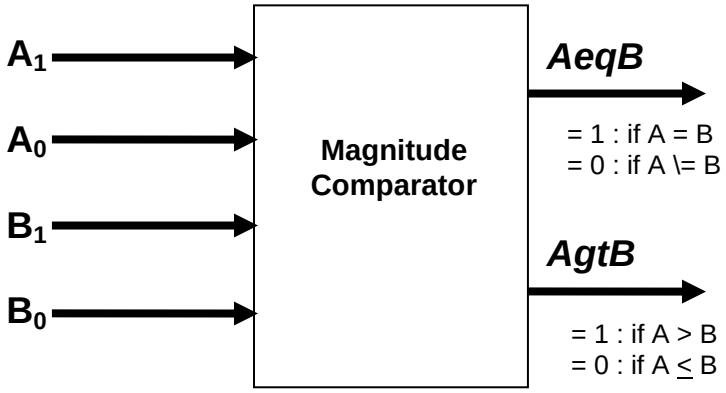
This could be remedied by making the signal assignment to **ReceivedParityOdd** ***a <u>Concurrent</u> assignment, by moving it outside of the process. (See following example)***

2) VHDL considers the conditional tests: if ((**ReceivedParityOdd** = '1') ***and*** elsif ((**ReceivedParityOdd** = '0') ***to be treating the Entity <u>OUTput</u> signal*** **ReceivedParityOdd** ***as an <u>"input"</u> to this statement; which it will not allow. This problem is most easily fixed by making a "temporary" <u>internal</u> version of this same signal (defining it locally in the Architecture). The following example shows this.***

```
architecture Behavioral of ParityChecker is
        signal ReceivedParityOdd_temp: STD_LOGIC;     -- temporary, internal version
begin

--Determine the parity of the received signal (Concurrently, OUTSIDE THE PROCESS)
ReceivedParityOdd_temp <= (Data(0) XOR Data(1) XOR Data(2) XOR Data(3) XOR ParityBit);

-- We need an internal (temp) version of this signal so that it can be used as an "input" to the
-- logic inside the PROCESS.
-- We also do need the "real" version of this signal as an Entity "OUTput" signal, too:
ReceivedParityOdd <= ReceivedParityOdd_temp;       -- Send answer out



parity_check: process (ReceivedParityOdd_temp, ExpectParityOdd) is
begin

        if ( (ReceivedParityOdd_temp = '1') AND (ExpectParityOdd = '1'))
                then ReceivedParityOK <= '1';  -- Expected Odd and received Odd parity

        elsif  ( (ReceivedParityOdd_temp = '0') AND (ExpectParityOdd = '0') )
                then ReceivedParityOK <= '1';  -- Expected Even and received Even parity

        else ReceivedParityOK <= '0';  -- Otherwise, there was a parity error
        end if;

end process parity_check;

end Behavioral;
```

2. **Design a logic gate circuit (not VHDL) for a 2-bit Magnitude Comparator to compare two positive binary numbers (A, B), each having only 2 bits ($A=A_1A_0$ and $B=B_1B_0$). The Comparator should output two signals: (*AeqB* meaning A=B) and (*AgtB* meaning A>B) to describe the relative magnitudes of the two numbers.**

You may use traditional minimization methods (K-maps?) to find a minimum sum-of-products or product-of-sums implementation of each output function.



SOLUTION:

The (AeqB) and (AgtB) outputs can be generated from a minimum Sum-of-Products or Product-of-Sums implementation, using A_0, A_1, B_0, B_1 as input variables (16 row truth table and 16 cell K-Map). The truth table would appear as:

A_1	A_0	B_1	B_0	A=B	A>B
0	0	0	0	1	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	1
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	1
1	0	0	1	0	1
1	0	1	0	1	0
1	0	1	1	0	0
1	1	0	0	0	1
1	1	0	1	0	1
1	1	1	0	0	1
1	1	1	1	1	0

AeqB (A=B)

A_1A_0 \ B_1B_0	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	0
10	0	0	0	1

$$(AeqB) = \overline{A_1} \bullet \overline{A_0} \bullet \overline{B_1} \bullet \overline{B_0} + \overline{A_1} \bullet A_0 \bullet \overline{B_1} \bullet B_0 + A_1 \bullet \overline{A_0} \bullet B_1 \bullet \overline{B_0} + A_1 \bullet A_0 \bullet B_1 \bullet B_0$$

4 AND Gates + 1 OR Gate + 4 Inverters = 9 Gates (24 Gate Inputs)

AeqB (A=B)

A_1A_0 \ B_1B_0	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	0
10	0	0	0	1

$$(AeqB) = (\overline{A_0} + B_0) \bullet (\overline{A_1} + B_1) \bullet (A_0 + \overline{B_0}) \bullet (A_1 + \overline{B_1})$$

4 OR Gates + 1 AND Gate + 4 Inverters = 9 Gates (<u>16</u> Gate Inputs)

AgtB (A>B)

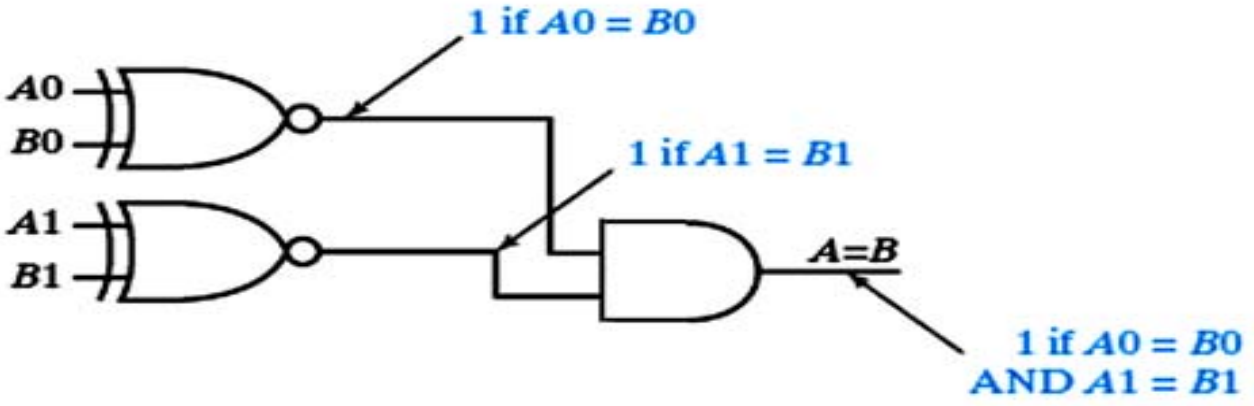
A_1A_0 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
10	1	1	0	0

$$(AgtB) = A_1 \bullet \overline{B_1} + A_0 \bullet \overline{B_1} \bullet \overline{B_0} + A_1 \bullet A_0 \bullet \overline{B_0}$$

3 AND Gates + 1 OR Gate + 2 Inverters = 6 Gates (13 Gate Inputs)

3. **Create a VHDL description of your 2-bit Magnitude Comparator <u>using a Data Flow (Concurrent) Architecture</u>. Use 2-bit vectors for the input signals A, B)**
   a. **Use your knowledge of XOR/XNOR functions to simplify the description of the (AeqB) function. (You may use logic similar to what we derived in lecture for the 4-bit comparator example.)**
   b. **Implement your best (AgtB) logic from Problem #2 as a Simple Concurrent Signal Assigment.**

The (AeqB) output can be simply generated using XNOR gates to test the equivalence of each of the two individual bit positions; with the XNOR gate outputs "AND"ed together (as was shown in lecture).




```
--------------------------------------------------------------------------------
-- Company: California Polytechnic State University
-- Engineer: Dr. Waynonius Pilkingtonium
-- Create Date:    23:50:34 03/01/2008
-- Design Name:
-- Module Name:    Comparator2Bit - DataFlow
-- Project Name:             Two-Bit Comparator
-- Target Devices: Nexys-2  Spartan 3E-500 FG320-4
-- Tool versions: ISE 9.2i
-- Description: Two-Bit Comparator using DataFlow Architecture
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Comparator2Bit is
   Port ( A : in  STD_LOGIC_VECTOR(1 downto 0);
          B : in  STD_LOGIC_VECTOR(1 downto 0);
        AeqB : out  STD_LOGIC;          -- True if A = B
        AgtB : out  STD_LOGIC);         -- True if A > B
end Comparator2Bit;
```

```
architecture DataFlow of Comparator2Bit is

begin

-- Hopefully, you arrived at these Boolean expressions for the 2-bit comparator:

AeqB <= ( A(1) xnor B(1) )  and  ( A(0) xnor B(0) );

AgtB <= ( A(1) and not B(1) )   or   ( A(0) and not B(1) and not B(0) )
                                     or   ( A(1) and A(0) and not B(0) );

end DataFlow;
```

4. Since most programmable logic devices do not have XOR or XNOR gates available inside for logic implementation, a VHDL compiler must substitute a SOP or POS realization for any XOR or XNOR functions. If this was done to your AeqB logic expression from your VHDL code in Part 3 above:

a. What would the resulting logic gate implementation created by the VHDL compiler look like? Write a Boolean expression for the logic that the compiler is likely to create for your AeqB output if it substitutes an SOP expression for any XOR or XNOR functions in your VHDL dataflow expression from 3a above?

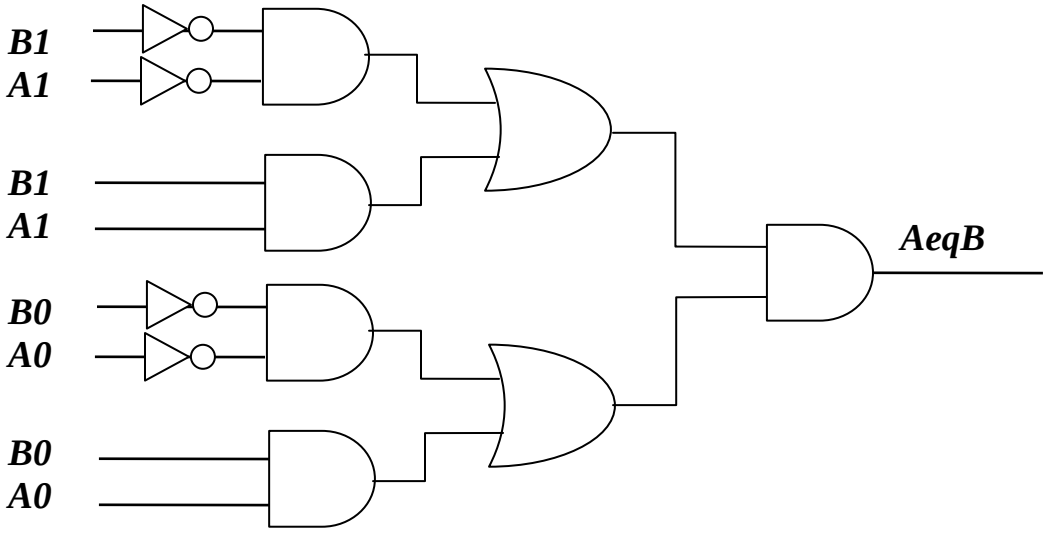
$$(AeqB) = \left(\overline{A_1 \oplus B_1}\right) \bullet \left(\overline{A_0 \oplus B_0}\right)$$

$$= \left[\left(\overline{A_1} \bullet \overline{B_1}\right) + \left(A_1 \bullet B_1\right)\right] \bullet \left[\left(\overline{A_0} \bullet \overline{B_0}\right) + \left(A_0 \bullet B_0\right)\right]$$ using an SOP version of XNOR

5 AND Gates + 2 OR Gates + 4 Inverters = 11 Gates (18 Gate Inputs)

b. Would this synthesized expression be better or worse (from a gate count perspective) than the best one you achieved with traditional methods?

If this solution was implemented exactly as shown it would be slightly worse in terms of the Gate Count and Gate Inputs than the best (POS) version of the "AeqB" output derived above with a K-Map. More importantly, notice that this solution is actually a <u>3-level</u> design (rather than a simple 2-level SOP or POS).



What the Xilinx VHDL compiler will actually create:

The initial VHDL compilation will substitute an SOP equivalent (a "macro") for any XOR functions, and will actually create a <u>3-level</u> circuit using another logic gate to handle both the inversion of the XOR function (to make it an XNOR) and the "AND" logic needed to combine the two individual XNOR results:

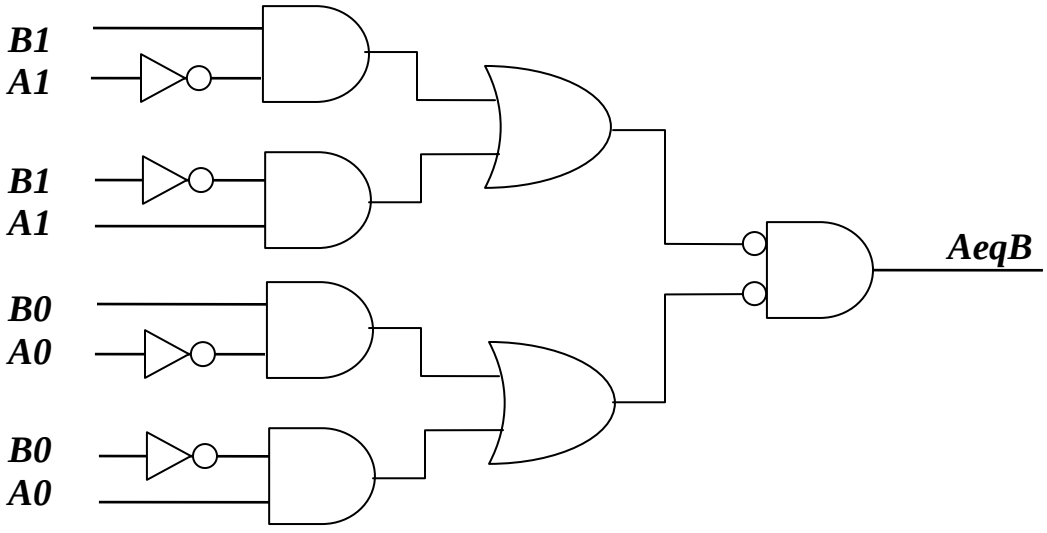
$$(AeqB) = \overline{(A_1 \oplus B_1)} \bullet \overline{(A_0 \oplus B_0)} = \left[not\left(A_1 \oplus B_1\right) \right] \bullet \left[not\left(A_0 \oplus B_0\right) \right]$$

$$= not\left[\left(\overline{A_1} \bullet B_1\right) + \left(A_1 \bullet \overline{B_1}\right) \right] \ AND \ not\left[\left(\overline{A_0} \bullet B_0\right) + \left(A_0 \bullet \overline{B_0}\right) \right] \text{ using an SOP version of XOR}$$

$$= \left[\left(\overline{A_1} \bullet B_1\right) + \left(A_1 \bullet \overline{B_1}\right) \right] \ nor \ \left[\left(\overline{A_0} \bullet B_0\right) + \left(A_0 \bullet \overline{B_0}\right) \right]$$

4 AND Gates + 2 OR Gate + 4 Inverters + 1 NOR Gate = 11 Gates (18 Gate Inputs)

Note that this again represents a <u>3-level circuit</u> (a 2-level SOP for each of the XOR functions, with their outputs combined into a NOR gate (3rd level).

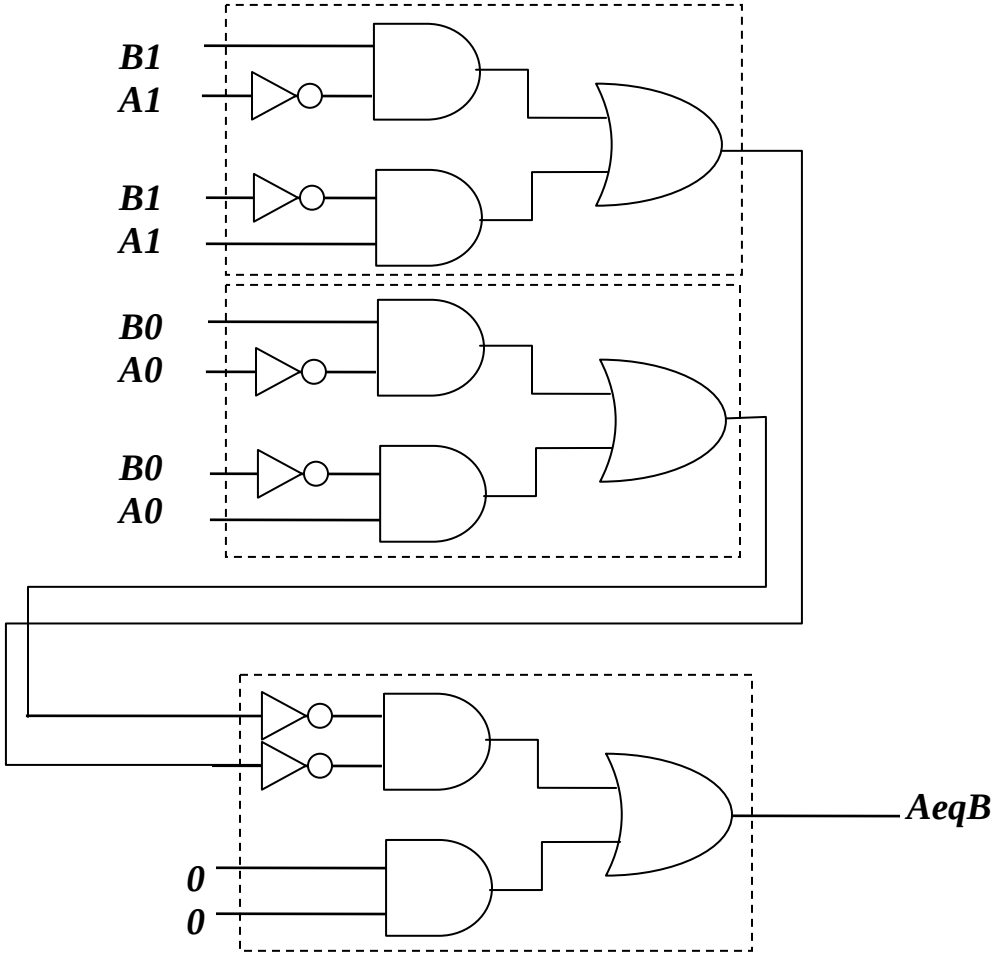


For the Spartan 3/3E FPGAs (like on the Nexys Boards), the configurable logic blocks (CLBs) can implement this type of logic (using its Look Up Tables). It will actually implement the original truth table (using something similar to a "Type 0 MUX" design)

Most PAL and CPLD devices have a 2-level internal gate structure, and so when the design is later implemented ("targeted") into a PAL or CPLD, this would have to either be further simplified to a 2-level equivalent (SOP or POS):

$$\begin{aligned}(AeqB) &= \left[\left(\overline{A_1}\bullet B_1\right)+\left(A_1\bullet\overline{B_1}\right)\right] \;\; nor \;\; \left[\left(\overline{A_0}\bullet B_0\right)+\left(A_0\bullet\overline{B_0}\right)\right] \\ &= \overline{\left[\left(\overline{A_1}\bullet B_1\right)+\left(A_1\bullet\overline{B_1}\right)\right]} \;\bullet\; \overline{\left[\left(\overline{A_0}\bullet B_0\right)+\left(A_0\bullet\overline{B_0}\right)\right]} \\ &= \left[\overline{\left(\overline{A_1}\bullet B_1\right)}\bullet\overline{\left(A_1\bullet\overline{B_1}\right)}\right]\bullet\left[\overline{\left(\overline{A_0}\bullet B_0\right)}+\overline{\left(A_0\bullet\overline{B_0}\right)}\right] \\ &= \left(A_1+\overline{B_1}\right)\bullet\left(\overline{A_1}+B_1\right)\bullet\left(A_0+\overline{B_0}\right)\bullet\left(\overline{A_0}\bullet B_0\right)\end{aligned}$$

Or (more likely) the output of the 2-level implementation of each of the XOR functions would have to be fed to another 2-level logic circuit block to handle the NOR combination; thus using up more logic resources than might have been used if the best POS version was coded in VHDL originally. (Compilers will do the best they can,…but sometimes how we describe our logic in VHDL will affect how efficiently it will be implemented in a programmable device; especially when constrained by the available internal logic structure of a programmable device.)



5. Create several VHDL descriptions of your 2-bit Magnitude Comparator <u>using Behavioral Models (with a Process)</u>. You need only specify the Architecture (assuming the Entity is the same as you specified above).

a. Version 1: Use the VHDL comparison operators (=, !=, >, <) on <u>single-bit signals</u> [Example: (A(0) = B(0))] and additional conditional logic to create the two needed outputs <u>using an IF-THEN-ELSE</u> statement for each. (This uses an algorithmic approach to the problem.)

<u>Solution:</u>

```
architecture Behavioral of comparator2bit  is
begin

ifcompare: process (A, B)  is
begin

    if ( A(1) = B(1) ) and ( A(0) = B(0) )
            then AeqB <= '1';
            else AeqB <= '0';
    end if;

    if (A(1) > B(1)) or ( ( A(1) = B(1)) and ( A(0) > B(0)) )
            then AgtB <= '1';
            else AgtB <= '0';
    end if;

end process ifcompare;

end Behavioral;
```

Alternative IF statements:

```
    if  ( A(1) > B(1) )
            then   AgtB <= '1';
    elsif  ( A(1) = B(1) ) and ( A(0) > B(0) )
            then   AgtB <= '1';
    else           AgtB <= '0';
    end if;
---------------------------------------------------------------
    if (A(1) > B(1))
    then   AgtB <= '1';
    else      if   ( A(1) = B(1) )
              then     if ( A(0) > B(0) )
                       then AgtB <= '1';
                       else AgtB <= '0';
                       end if;
              else AgtB <= '0';
              end if;
    end if;
```

b. Version 2: Use one or two CASE statements to produce the two needed outputs.

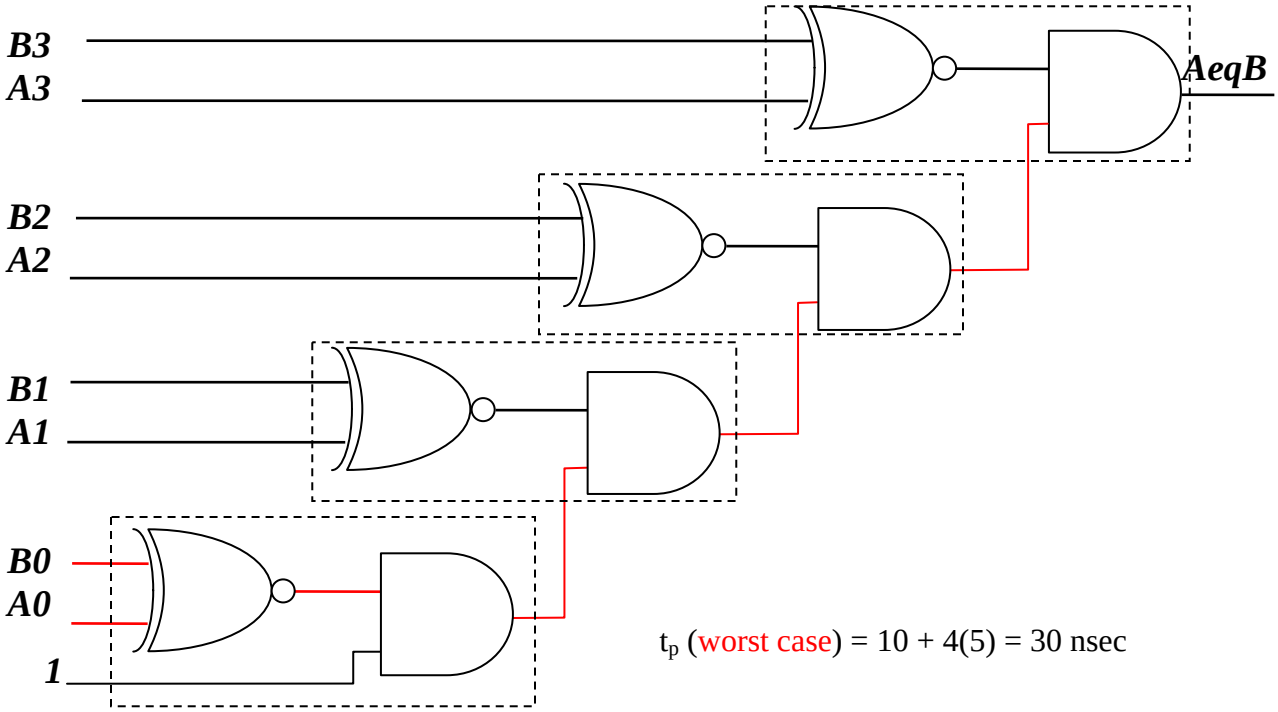
Solution:

```
architecture Behavioral of comparator2bit  is
signal AB: STD_LOGIC_VECTOR (3 downto 0);
begin

AB <= A & B;   --concatenate A,B into a 4-bit vector to use as a CASE selector

casecomp: process (AB) is
begin
    case (AB) is
            --  Check for cases when A = B
            when "0000" | "0101" | "1010" | "1111"
                                        => AeqB<= '1';  AgtB <= '0';

            --  Check for cases when A > B
            when "1110"  | "1101"  | "1100"
                         | "1001" | "1000"
                                  | "0100"
                                        => AeqB<= '0';  AgtB <= '1';

            --  All other cases:  A < B
            when others                 => AeqB<= '0';  AgtB <= '0';
    end case;
end process casecomp;

end Behavioral;
```

NOTE: This version will probably synthesize more complex logic than the previous versions of VHDL code, since this description is more like identifying most of the lines in a Truth Table. *(*The synthesized logic will be more like a *canonical form* than a *reduced form* of the logic.)

6. Figure 6-77 in Wakely shows an "iterative circuit" approach to creating a multiple-bit comparator. It is similar but subtly different from the 4-bit comparator hardware we derived in class.

c. From this Figure, draw the gate-level circuit diagram for an <u>iterative circuit implementation</u> of a 4-bit comparator (tests if A=B, where A = $A_3A_2A_1A_0$ and B= $B_3B_2B_1B_0$) and outputs a single signal *EQ* = 1 when A=B)

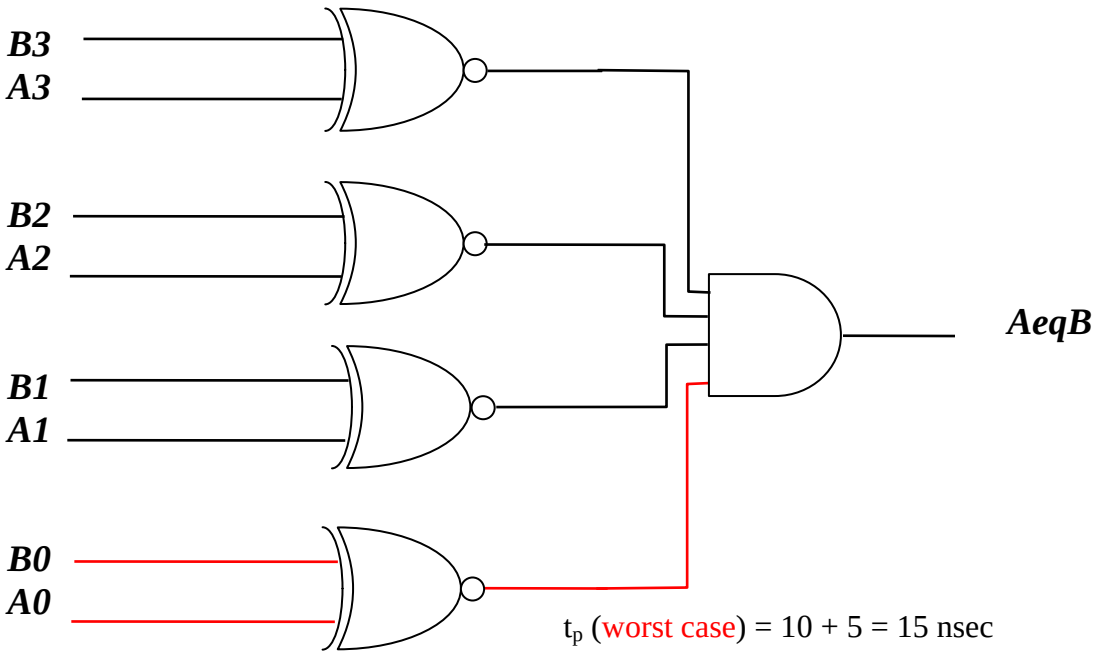


t_p (worst case) = 10 + 4(5) = 30 nsec

d. Determine the worst case propagation delay for this iterative circuit, if the logic gates have the following propagation delays:

Gate:	AND	XNOR
t_{pHL}	5 nsec	10 nsec
t_{pLH}	5 nsec	10 nsec

e. Using these same values for gate delays, determine the worst case propagation delay for the version of the 4-bit comparator circuit derived in class.



t_p (worst case) = 10 + 5 = 15 nsec