Sorter-Absorber Theory

Suppose we are given an array of Sum-Of-Products containing all possible covers for a particular function.

Goals:

1. Sort Minterms by their size.
   - Bubble sort, parallel in 4 rows. Two columns implementing even and odd sorting. Data is moved from one memory to another by increments of 4. After a pass through sorters, data is sent back from second memory to first.
2. Absorb all possible minterms.
   Since data may be pre-sorted, to guarantee full absorption every data needs to be compared to every other data. Therefore after sorting occurred, an upside-down sort is performed.

Law of Absorption:
A+AB = A   more complex form ABC!D + ABC = ABC
Example: K-map

AB \ CD	00	01	11	10
00	0	1	1	0
01	1	1	1	1
11	0	0	0	0
10	0	0	1	0

Lets suppose that the program generates all possible cubes for this cover:
!A!B!CD + !A!BCD + !AB!C!D + !AB!CD + !ABCD + !ABC!D + A!BCD +
!A!BD + !AB!C + !A!CD + !ACD + !ABC + !BCD + !AB + !AD.
Every small cube, such as !A!B!CD can be absorbed by a bigger cube, such as !AB!C or even !AB. The output that we are looking to get for this particular example from our circuit is: !AB + !AD +!BCD. (*Note the bigger cubes are on top, because output is sorted*)

Problem size: sort terms of up to 16 variables, up to 64k terms.

Since a variable may be missing or term may be absorbed there are four possible states a variable may be in:

Encoding	State Meaning
"00"	1 – indicates that the variable is present and asserted high (A)
"01"	0 – indicates that the variable is present and asserted low (!A)
"10"	X – indicates that the variable is not present – don't care
"11"	? -- indicates that the value has been absorbed

If the input word is four variable wide, product term !ABC!D would become "10 11 11 10" and AC would be encoded as "11 01 11 01".

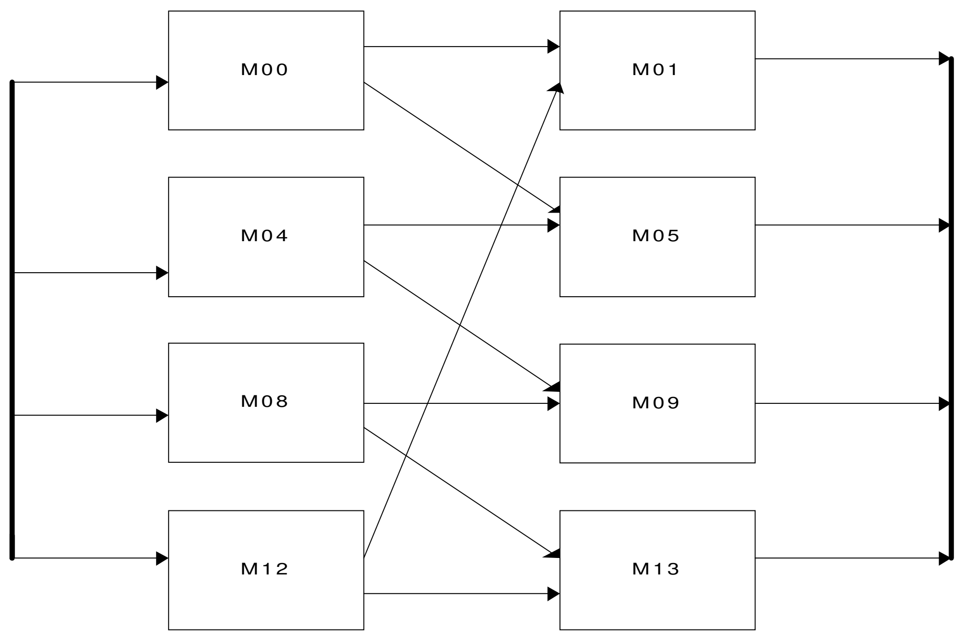
Given two terms and comparing them variable by variable, we can state that:

1) If both of the product terms contain missing variables in the same position or if they contain same values in those positions, then they are the same and therefore both can be absorbed into each other.

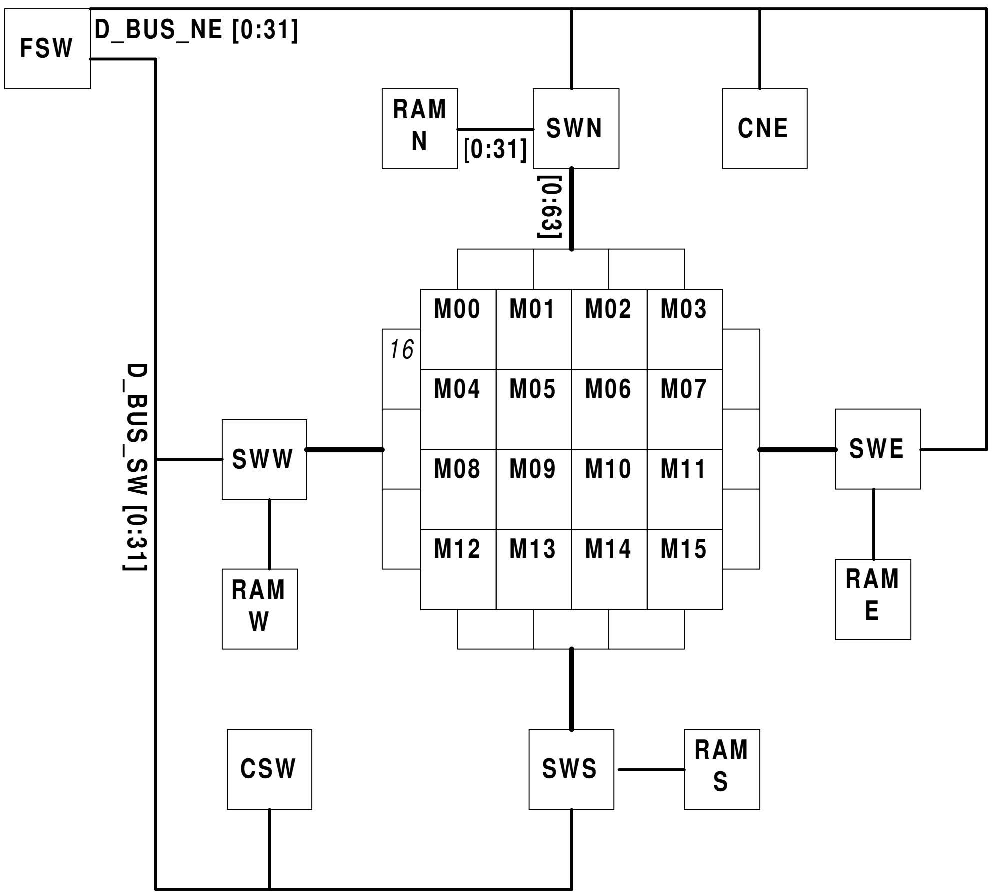
2) Given the rest of the variables are equal, if one of the variables is X, and same variable in the other term is either 1 or 0, then the product containing X absorbs other product. This absorption decision is "recorded" by either state of the state machine or carry of the iterative circuit. Also, there could not be previous decision made to perform absorption in the opposite way.

3) If one of the variables being compared are opposite and asserted (either 1 or 0 and another variable is opposite and 0 or 1), the products are different and absorption cannot take place between the given two terms.

Sorting Scheme



DEC-Perle Architecture



Summary of DEC-Perle board:
4 by 4 Matrix of 3090 FPGAs
4 Programmable Matrix Switches
4 SRAM memories
2 Controllers
10 bit wide bus for signals.
Each chip in array connected to its neighbors by 16 bit bus, and to four 16 bit memory buses.

Pipeline Design

Reason for Pipelining: To be able to use large terms (16 variables) with only 16 bit memory bus connection. Greatly complicates design by distributing memory loads and writes over four clock cycles.

Pipeline Cycles:

Processor	Cycle **0** init_0	1 init_1	2 init_2	3 init_3	4 init_4	5 init_5	6 init_6	7 init_7
M00	Read n-1	L *l* A_0	L *h* A_0	L *l* B_0	L *h* B_0	Sort(A_0, B_0)	W B_0	W A_0
M04		L *l* B_1	L *h* B_1	L *l* A_1	L *h* A_1	Sort(A_1, B_1)	W B_1	W A_1
M08		No-op	L *l* A_2	L *h* A_2	L *l* B_2	L *h* B_2	Sort (A_2, B_2)	W B_2
M12		No-op	L *l* B_3	L *h* B_3	L *l* A_3	L *h* A_3	Sort(A_3, B_3)	W B_3
M01		No-op	No-op	No-op	No-op	No-op	L B_0	W B_0, L A_0
M05		No-op	No-op	No-op	No-op	No-op	L B_1	W B_1 L A_1, B_0
M09		No-op	No-op	No-op	No-op	No-op	No-op	L B_2, L B_1
M13		No-op	No-op	No-op	No-op	No-op	No-op	L B_3
CSW	R A_0 B_1	R A_2, B_3	RA_1, B_0	R A_3, B_2				R A_4, B_5
CNE								

Figure 1. Pipeline Initialization

8 sust_0	9 sust_1	10 sust_2	11 sust_3	12 sust_4	13 sust_5	14 sust_6
L l A_4	L h A_4	L l B_4	L h B_4	Sort (A_4, B_4)	W B_4	W A_4
L l B_5	L h B_5	L l A_5	L h A_5	Sort(A_5, B_5)	W B_5	W A_5
W A_2	L l A_6	L h A_6	L l B_6	L h B_6	Sort (A_6, B_6)	W B_6
W $A_{\square}$	L l B_7	L h B_7	L l A_7	L h A_7	Sort(A_7, B_7)	W B_7
Hold A_0 L B_3	WB l A_0	WB h A_0	WB l B_3	WB h B_3	L B_4	L A_4 W B_4
Sort (A_1, B_0)	WB l B_0	WB h B_0	WB l A_1	WB h A_1	L B_5	L A_5, B_4 W B_5
W B_2, L A_2	Sort (A_2, B_1)	WB l A_2	WB h A_2	WB l B_1	WB h B_1	L B_5, B_6
W B_3 , L A_3, L B_2	Sort (A_3, B_2)	WB l B_2	WB h B_2	WB l A_3	WB h A_3	L B_7
R A_6, B_7	R A_5, B_4	R A_7, B_6				R A_8 B_9
			W A_0, B_0	W A_2, B_2	WB$_3$, A_1	W B_1, A_3

Figure 2. Sustained Pipeline Operation

Processor/ Cycle	**29** final_0	30 final_1	31 final_2	32 final_3	33 final_4	34 final_5	**35** final_6
M00	No-op	No-op	No-op	No-op	No-op	No-op	No-op
M04	No-op	No-op	No-op	No-op	No-op	No-op	No-op
M08	W A_{14}	No-op	No-op	No-op	No-op	No-op	No-op
M12	W A_{15}	No-op	No-op	No-op	No-op	No-op	No-op
M01	Hold A_{12} L B_{15}	WB *l* A_{12}	WB *h* A_{12}	WB *l* B_{15}	WB *h* B_{15}	No-op	No-op
M05	Sort (A_{13}, B_{12})	WB *l* B_{12}	WB *h* B_{12}	WB *l* A_{13}	WB *h* A_{13}	No-op	No-op
M09	W B_{14}, L A_{14}	Sort (A_{14}, B_{13})	WB *l* A_{14}	WB *h* A_{14}	WB *l* B_{13}	WB *h* B_{13}	No-op
M13	W B_{15} , L A_{15}	Sort (A_{15}, B_{14})	WB *l* B_{10}	WB *h* B_{10}	WB *l* A_{11}	WB *h* A_{15}	No-op
CSW							
CNE				W A_{12}, B_{12}	W A_{14}, B_{14}	WB$_{15}$, A_{13}	W B_{13}, A_{15}

Figure 3. Pipeline draining

Design of Controllers, Switches and Sorters

The code for these was developed using the pipeline diagrams developed earlier. There are six different sorter configurations based on different tasks performed on same clock cycles. All sorters use same hierarchical sorting network, just the timing needs to be rewritten.

VHDL Code structure:

Hierarchy of Design:
1. Top level board schematic
    2. Controllers and SRAM, FPGA Array
        3. FPGA <- sorter circuit schematics
                4.Sorting Iterative Network
                    5. Iterative Circuit Cell Description

CODE SAMPLES: Due to size of code (7+ pages of code) only one sorter configuration is shown here

Example 1: Iterative cell description:

```
--
--*****************************************************
--* Pavel Kashubin
--* EE574 FP
--* Rev 1.1          5/23/98
--*****************************************************
-- This is a code for the cell of the interative absorb/compare circuit.
-- This cell is called from the above level and can be repeated
-- as many times as necessary.
--
--******************* Signal descriptions: ****************************
-- A - input term of the first word.
-- B - input term of the second word.
-- prev_state  - iterative signal propagated from previous cell. Initial
-- cell must have 11 as the input.
-- state_next - output of the cell to the next cell of the decoder.
-- comp_prev_state - input from previus cell. Initial input is 000, or
-- A=B condiition.
-- comp_next-state - output from cell to the next cell or decoder.
-- Note: since data type four_val is not supported by synthesis tool,
-- the A and B  replaced by the bit_vector with encoding
-- '?' - "00" , 'X' - "01", 0 - "10" , 1 - "11"

library ieee;
use ieee.std_logic_1164.all;
--use work.data_types.all;

entity cell is
port(
        A:          in std_logic_vector(1 downto 0);
        B:          in std_logic_vector(1 downto 0);
        prev_state:           in std_logic_vector(1 downto 0);
        state_next:           out std_logic_vector(1 downto 0);
        comp_prev_state : in std_logic_vector(2 downto 0);
        comp_next_state : out std_logic_vector(2 downto 0));
end cell;

architecture arch_cell of cell is

-- signals state_next and prev_state are encoded as follows:
-- 11 - both a and b can be absorbed into each other. Also implies that
-- they are equal.
-- 10 A can be absorbed into B
-- 01 B can be absorbed into A
-- 00 No absorbtion can take place

begin
state_process: process (prev_state, A,  B)
begin
case prev_state is
        when "11" =>
                if ((A = B) and (A = "10" or A = "11" or A = "01"))
                then state_next <= "11";
                elsif (B = "01" and (A ="10" or A="11")) then
                state_next <="10";
                elsif (A = "01" and (B = "10" or B = "11")) then
                state_next <= "01";
                else state_next <= "00";
                end if;
        when "10" =>
                if ((B = A or B = "01") and (A = "10" or A = "11"))
                then state_next <= "10";
                else state_next <= "00";
                end if;
        when "01" =>
                if ((A = B or A = "01")  and (B = "10" or B = "11"))
                then state_next <= "01";
                else state_next <= "00";
                end if;
        when "00" =>
                 state_next <= "00";
        when others => state_next <= "00";
        end case;
 end process;
-- Signals comp_prev_state and comp_next_state are encoded as follows:
-- 000 - A=B
-- 001 - A>B
```

```
-- 010 - 1BX
-- 011 - 2BX
-- 100 - A<B
-- 101 - 1AX
-- 110 - 2AX

compare: process (A, B, comp_prev_state)
begin
 case comp_prev_state is
        when "000" =>
                if (A = "11" and B = "10") then
                 comp_next_state <= "001";
                elsif (A= "10" and B="11") then
                 comp_next_state <= "100";
                elsif (B="01" and A /="01") then
                 comp_next_state <= "010"; -- go to 1BX
                elsif (B/="01" and A = "01") then
                 comp_next_state <= "101"; -- go to 1AX
                else
                 comp_next_state <= "000";
        end if;
    when "001" =>
                if (A /= "01" and B = "01") then
                comp_next_state <= "010";
                elsif (A = "01" and  B /= "01") then
                comp_next_state <= "101";
                else
                comp_next_state <= "001";
                end if;
        when "010" =>        -- 1BX
                if (A = "01" and  B /= "01") then
                 comp_next_state <= "101";
                elsif (A /= "01" and B = "01") then
                comp_next_state <= "011";
                else comp_next_state <= "010";
                end if;
        when "011" =>  -- 2BX
                if (A = "01" and  B /= "01")then
                comp_next_state <= "010";
                else comp_next_state <= "011";
                end if;
        when "100" => -- A < B
                if (A /= "01" and B = "01") then
        comp_next_state <= "010";
        elsif (A = "01" and B /= "01") then
        comp_next_state <= "101";
        else
        comp_next_state <= "100";
        end if;
        when "101" =>  -- 1AX
                if (A ="01" and B /="01") then
        comp_next_state <= "110"; --go to 2AX
        elsif (A /= "01" and B ="01") then
        comp_next_state <= "010"; -- go to 1BX
        else comp_next_state <= "101";
                end if;
        when "110" =>   -- 2AX
                if (A /= "01" and B = "01" ) then
        comp_next_state <= "101"; --go to 1AX
        else comp_next_state <= "110"; -- stay ihn 2AX
        end if;
        when others => comp_next_state <= "000";
        end case;
end process compare;
end arch_cell;
```

Example 2: Sorter Configuration

```
-- This file describes architecture of sorter circuit as it will be implemented in an FPGA
chip. This file will be modified for a different
-- configurations of sorter. Inputs are the 32 bit words, but will need to be modified to
16 bit words because bus cycle will be used.
-- Command is the 9 bit ring bus that is used for signalling. One bit is used as a signal
if sorting happened.
-- Configuration 1 (M00 and M04)

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

entity sorter_circuit_1 is port (
Clock : in std_logic;
MBusW: in std_logic_vector (15 downto 0);          -- Horizontal MBusW used
for input
MBusN : inout std_logic_vector (15 downto 0);          -- 16 bits Vertical Bus
North
MBusS : inout std_logic_vector (15 downto 0);          -- 16 bits Vertical Bus
South
MBusE: inout std_logic_vector (15 downto 0);       -- Horizontal MBusE output
```

```
DCH_L : inout std_logic_vector ( 15 downto 0);                    -- 16 bits
Direct connect Horizontal LEFT
DCH_R : inout std_logic_vector ( 15 downto 0);                    -- 16 bits
Direct connect Horizontal RIGHT
DCV_T : inout std_logic_vector ( 15 downto 0);                    -- 16 bits
Direct connect Horizontal TOP
DCV_B:  inout std_logic_vector (15 downto 0);                -- 16 bits direct
connect Vertical Bottom
CYCLE: inout std_logic_vector (4 downto 0);                     -- Ring Matrix
signals
Noop: inout std_logic;
Reset: in std_logic


); END sorter_circuit_1;

--16 bits MBusW oused for output.

architecture config_1 of sorter_circuit_1 is

-- COMPONENTS
-- declare sorter schematic component here.
COMPONENT SCHEMATIC2 PORT  (
        prev_state : IN std_logic_vector(1 DOWNTO 0);
        A : IN std_logic_vector(31 DOWNTO 0);
        B : IN std_logic_vector(31 DOWNTO 0);
        state_next : OUT std_logic_vector(1 DOWNTO 0);
        comp_prev_state : IN std_logic_vector(2 DOWNTO 0);
        comp_next_state : OUT std_logic_vector(2 DOWNTO 0)
); END COMPONENT;

--
-- Use this to declare enumerated types for command. it will help to make program
more readable
-- Signal declarations
SIGNAL A:  std_logic_vector (31 downto 0);
SIGNAL B:  std_logic_vector (31 downto 0);
SIGNAL AN:  std_logic_vector (31 downto 0);
SIGNAL BN:  std_logic_vector (31 downto 0);
SIGNAL final_abs : std_logic_vector (1 downto 0);   -- fianl absorbtion  output.
SIGNAL final_comp: std_logic_vector (2 downto 0); -- fianl comparison output
SIGNAL enable : std_logic; -- used to signal another process when to execute
-- Constants are declared for easier reading
CONSTANT init_0: std_logic_vector (4 downto 0) := "00000";
CONSTANT init_1: std_logic_vector (4 downto 0) := "00001";
CONSTANT init_2: std_logic_vector (4 downto 0) := "00010";
CONSTANT init_3: std_logic_vector (4 downto 0) := "00011";
CONSTANT init_4: std_logic_vector (4 downto 0) := "00100";
CONSTANT init_5: std_logic_vector (4 downto 0) := "00101";
CONSTANT init_6: std_logic_vector (4 downto 0) := "00110";
CONSTANT init_7: std_logic_vector (4 downto 0) := "00110";
CONSTANT sust_0: std_logic_vector (4 downto 0) := "00111";
CONSTANT sust_1: std_logic_vector (4 downto 0) := "01000";
CONSTANT sust_2: std_logic_vector (4 downto 0) := "01001";
CONSTANT sust_3: std_logic_vector (4 downto 0) := "01010";
CONSTANT sust_4: std_logic_vector (4 downto 0) := "01011";
CONSTANT sust_5: std_logic_vector (4 downto 0) := "01100";
CONSTANT sust_6: std_logic_vector (4 downto 0) := "01101";
CONSTANT final_0: std_logic_vector (4 downto 0) := "01110";
CONSTANT final_1: std_logic_vector (4 downto 0) := "01111";
CONSTANT final_2: std_logic_vector (4 downto 0) := "10000";
CONSTANT final_3: std_logic_vector (4 downto 0) := "10001";
CONSTANT final_4: std_logic_vector (4 downto 0) := "10010";
CONSTANT final_5: std_logic_vector (4 downto 0) := "10011";
CONSTANT final_6: std_logic_vector (4 downto 0) := "10100";
CONSTANT idle: std_logic_vector(4 downto 0) := "11111";
SIGNAL cps : std_logic_vector (2 downto 0);
SIGNAL  ps : std_logic_vector ( 1 downto 0);
--
--ALIAS NOOP: std_logic is Command (9);
--ALIAS RESET: std_logic is Command(8);
-- THESE ALIASES MUST BE CHANGED DEPENDING ON SORTER CONFIG.


BEGIN
-- Create an instance of sorting network and call it with appropriate parameters for
needed configuration
-- This is configuration 1 (M00 and M04)
ps <= "11"; cps <= "000";
sort: SCHEMATIC2 port map  ( ps, A, B, final_abs, cps, final_comp);
-- This process controls the outputs based on cycle inputs received from controller
exec_command:  process (cycle, MBusW, BN, AN)
begin
enable <= '0';
case cycle is
  when init_0 =>
  -- Controller will read memory - nothing to do in this cycle.
  when init_1 =>
        A(15 downto 0) <= MBusW;    -- read A_low from memory bus
        -- high impedance unused buses
        MBusN <= (others => 'Z');
        MBusS <= (others => 'Z');
```

```
            MBusE <= (others => 'Z');
            DCH_L <= (others => 'Z');
            DCV_T <= (others => 'Z');
            DCV_B <= (others => 'Z');
when init_2 =>
            A(31 downto 16) <= MBusW;
            -- high impedance
            MBusN <= (others => 'Z');
            MBusS <= (others => 'Z');
            MBusE <= (others => 'Z');
            DCH_L <= (others => 'Z');
            DCV_T <= (others => 'Z');
            DCV_B <= (others => 'Z');
  when init_3 =>
            B(15 downto 0) <= MBusW;
            -- high impedance
            MBusN <= (others => 'Z');
            MBusS <= (others => 'Z');
            MBusE <= (others => 'Z');
            DCH_L <= (others => 'Z');
            DCV_T <= (others => 'Z');
            DCV_B <= (others => 'Z');
when init_4 =>
            B(31 downto 16) <= MBusW;
            -- high impedance
            MBusN <= (others => 'Z');
            MBusS <= (others => 'Z');
            DCH_L <= (others => 'Z');
            DCV_T <= (others => 'Z');
            DCV_B <= (others => 'Z');
when init_5 =>
            -- enable sort
            enable <= '1';
            -- high impedance
            MBusE <= (others => 'Z');
            MBusN <= (others => 'Z');
            MBusS <= (others => 'Z');
            DCH_L <= (others => 'Z');
            DCH_R <= (others => 'Z');
            DCV_T <= (others => 'Z');
            DCV_B <= (others => 'Z');

when init_6 =>
    DCH_R <= BN(15 downto 0);
            MBusE <= BN(31 downto 16);
            -- high impedance
            MBusN <= (others => 'Z');
            MBusS <= (others => 'Z');
            DCH_L <= (others => 'Z');
            DCV_T <= (others => 'Z');
            DCV_B <= (others => 'Z');
when init_7 =>
    DCH_R <= AN(15 downto 0);
    MBusE <= AN(31 downto 16);
    -- high impedance unused buses
            MBusN <= (others => 'Z');
            MBusS <= (others => 'Z');
            MBusE <= (others => 'Z');
            DCH_L <= (others => 'Z');
            DCV_T <= (others => 'Z');
            DCV_B <= (others => 'Z');
when sust_0 =>
            A(15 downto 0) <= MBusW;
            -- high impedance unused buses
            MBusN <= (others => 'Z');
            MBusS <= (others => 'Z');
            MBusE <= (others => 'Z');
            DCH_L <= (others => 'Z');
            DCV_T <= (others => 'Z');
            DCV_B <= (others => 'Z');
  when sust_1 =>
            A(31 downto 16) <= MBusW;
    -- high impedance unused buses
            MBusN <= (others => 'Z');
            MBusS <= (others => 'Z');
            MBusE <= (others => 'Z');
            DCH_L <= (others => 'Z');
            DCV_T <= (others => 'Z');
            DCV_B <= (others => 'Z');
  when sust_2 =>
            B(15 downto 0) <= MBusW;
            -- high impedance unused buses
            MBusN <= (others => 'Z');
            MBusS <= (others => 'Z');
            MBusE <= (others => 'Z');
            DCH_L <= (others => 'Z');
            DCV_T <= (others => 'Z');
            DCV_B <= (others => 'Z');
when sust_3 =>
            B(31 downto 16) <= MBusW;
            -- high impedance unused buses
            MBusN <= (others => 'Z');
```

```
            MBusS <= (others => 'Z');
            DCH_L <= (others => 'Z');
            DCV_T <= (others => 'Z');
            DCV_B <= (others => 'Z');
when sust_4 =>
             -- enable sort
            enable <= '1' ;
            -- high impedance unused buses
            MBusE <= (others => 'Z');
            MBusN <= (others => 'Z');
            MBusS <= (others => 'Z');
            DCH_R <= (others => 'Z');
            DCH_L <= (others => 'Z');
            DCV_T <= (others => 'Z');
            DCV_B <= (others => 'Z');
when sust_5 =>
            DCH_R <= BN(15 downto 0);
            MBusE <= BN(31 downto 16);
            -- high impedance unused buses
            MBusN <= (others => 'Z');
            MBusS <= (others => 'Z');
            DCH_L <= (others => 'Z');
            DCV_T <= (others => 'Z');
            DCV_B <= (others => 'Z');
when sust_6 =>
      DCH_R <= BN(15 downto 0);
            MBusE <= BN(31 downto 16);
            -- high impedance
            MBusN <= (others => 'Z');
            MBusS <= (others => 'Z');
            DCH_L <= (others => 'Z');
            DCV_T <= (others => 'Z');
            DCV_B <= (others => 'Z');

when final_0 =>
    DCH_R <= A(15 downto 0);
    MBusE <= A(31 downto 16);
            -- high impedance unused buses
            MBusN <= (others => 'Z');
            MBusS <= (others => 'Z');
            DCH_L <= (others => 'Z');
            DCV_T <= (others => 'Z');
            DCV_B <= (others => 'Z');

when others =>
-- No-op for all other cycles final 1 through 6 in this configuration.
-- Buses and Direct connections are tri-stated.
MBusE <= (others => 'Z');
MBusN <= (others => 'Z');
MBusS <= (others => 'Z');
DCH_L <= (others => 'Z');
DCH_R <= (others => 'Z');
DCV_T <= (others => 'Z');
DCV_B <= (others => 'Z');
   -- No-op for all other cycles final 1 through 6 in this configuration.
   -- Buses and Direct connections are tri-stated.

end case;
end process;


select_output : process ( final_abs, final_comp, ENABLE)
variable MIN_W, MAX_W : std_logic_vector (31 downto 0); -- variables inside the loop
            begin
            if (ENABLE = '1') then
            if (final_abs = "11") then      -- if both A and B can be absorbed,
             for i in 0 to 31 LOOP     -- lets absorb A and zero out numbers
              MIN_W(i) := '0';
             end LOOP;
            AN <=  MIN_W;
            BN <= B;
              noop <= '0'; -- absorption took place
            elsif (final_abs = "10") then   -- if A is absorbed into B
              for i in 0 to 31 LOOP
        MIN_W(i) := '0';
       end LOOP;
              AN  <= MIN_W;
              BN <= B;                              -- B goes to BN output
              noop <= '0'; -- absorption occured
      elsif (final_abs = "01") then   -- B is absorbed into A
              for i in 0 to 31 LOOP
        MIN_W(i) := '0';
              end LOOP;
              AN <= MIN_W;
      BN <= B;           -- A goes to BN output
        noop <= '0'; -- absorption
            elsif (final_abs = "00") then
             if (final_comp = "001" or final_comp = "010" or final_comp ="011")
             then                           -- A > B
                AN <=  B;
                BN <= A;
                noop <= '0';  -- sorting took place
```

```
        elsif (final_comp = "100" or final_comp = "110" or final_comp ="101"
                or final_comp = "000") then    -- A < B or A=B=?
                    AN <= A;
          BN <= B;
                    noop <= 'Z';    -- no operation took place
        end if;
               end if;
               else
               end if;
               end process;
end config_1;
```