C. APPENDIX – C: VHDL CODE FOR THE MAIN DATA PATH AND DETAILED MEMORY MAP FOR THE DOWNLOADED PARAMETERS

C.1: MAIN MEMORY DECODER

```
------------------------------------------------------------------------
--  This block decodes the main memory space
------------------------------------------------------------------------
-- The initialization of the standard library files
------------------------------------------------------------------------
--This revision 2B to modify the test lut data to 18 bits
--Shweta Lolage: 10/05/2000
------------------------------------------------------------------------
library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
------------------------------------------------------------------------entity
main_memory_decoder is

port (clock,reset: in std_logic;
      main_address:in std_logic_vector(14 downto 0);
      main_data_in: in std_logic_vector(31 downto 0);
      main_select,main_read,main_write: in std_logic;
      monitor_data_in,misc_parameters_data_in:in std_logic_vector(31 downto 0);
      gainoffset_data_in: in std_logic_vector(7 downto 0);
      roadlut_data_in : in std_logic_vector (21 downto 0);
      testlut_data_in: in std_logic_vector(17 downto 0);
      registered_data: out std_logic_vector(31 downto 0);
      registered_address: out std_logic_vector(14 downto 0);
      monitor_enable,misc_parameters_enable,
      gainoffset_enable,testlut_enable,roadlut_enable : out std_logic;
      global_write_not_read,global_dataoutput_enable: out std_logic;
      main_data_out: out std_logic_vector(31 downto 0)
     );
end entity main_memory_decoder;
------------------------------------------------------------------
--Architecture body

architecture logic of main_memory_decoder  is
--
-- define memory map for address bits 14 downto 12
--
constant monitor_address_space:std_logic_vector(2 downto 0) := "0000";
constant misc_address_space: std_logic_vector(2 downto 0) := "0001";
constant gainoffset_address_space0: std_logic_vector(2 downto 0) := "0010";
constant gainoffset_address_space1: std_logic_vector(2 downto 0) := "0011";
constant gainoffset_address_space2: std_logic_vector(2 downto 0) := "0100";
constant gainoffset_address_space3: std_logic_vector(2 downto 0) := "0101";
constant test_address_space: std_logic_vector(2 downto 0) := "0110";

--- define constants for zero's to increase bit width of above memory space
constant zero8: std_logic_vector(7 downto 0) := "00000000";
constant zero10: std_logic_vector(9 downto 0) := "0000000000";
constant zero14: std_logic_vector(13 downto 0) := "00000000000000";
constant zero15: std_logic_vector(14 downto 0) := "000000000000000";
constant zero24: std_logic_vector(23 downto 0) := "000000000000000000000000";
constant zero23: std_logic_vector(22 downto 0) := "00000000000000000000000";
constant zero32 : std_logic_vector(31 downto 0) :=
"00000000000000000000000000000000";

--
--   Define constants to determine if this is a read or write mode
```

```
--
constant read_mode: std_logic := '0';
constant write_mode: std_logic := '1';
constant disabled: std_logic := '0';
constant enabled: std_logic := '1';


---  define signals to latch all external inputs
signal nmain_address,pmain_address: std_logic_vector(14 downto 0);
signal nmain_data_in,pmain_data_in: std_logic_vector(31 downto 0);
signal nmain_read,pmain_read,nmain_write,pmain_write,nmain_select,pmain_select:
std_logic;


 begin

--  This process block sets the enables for each address space if this smt
channel has been selected
process(main_address,pmain_select)
    variable address: std_logic_vector(3 downto 0);
    variable vmonitor,vmisc,vgainoffset,vtestlut,vroadlut: std_logic;
    variable vdata_out: std_logic_vector(31 downto 0);
    variable memory_area_select : std_logic;
-- this signal is to differentiate between the memory for memories on the chip
-- and the road data memory outside the chips

   begin
-- set default outputs

     vmonitor := '0';
     vmisc    := '0';
     vgainoffset := '0';
     vtestlut := '0';
     vroadlut := '0';
     vdata_out := zero32;

 address := main_address(12 downto 10);
  -- only need top 4 bits for address decoding
  -- Dr. Perry I have made this change according to the memory map we discussed

 memory_area_select := main_address(13);
--  if select line was high during last cycle then immediately enable
appropriate address space

    if(pmain_select = '1') then

       if (memory_area_select = '0') then
        case address is
         when monitor_address_space =>
            vmonitor := '1';
            vdata_out := monitor_data_in;

         when misc_address_space =>
            vmisc := '1';
            vdata_out := misc_parameters_data_in;

         when gainoffset_address_space0|gainoffset_address_space1 =>
            vgainoffset := '1';
            vdata_out := zero24&gainoffset_data_in;

         when gainoffset_address_space2|gainoffset_address_space3 =>
            vgainoffset := '1';
            vdata_out := zero24&gainoffset_data_in;
```

```
        when test_address_space =>
            vtestlut := '1';
            vdata_out := zero14&testlut_data_in;

      when others =>
               null;
             end case;

else
       vroadlut := '1';
       vdata_out := zero10&roadlut_data_in;
end if;

end if;

     monitor_enable <= vmonitor;
     misc_parameters_enable  <= vmisc;
     gainoffset_enable    <= vgainoffset;
     testlut_enable   <= vtestlut;
     roadlut_enable  <= vroadlut;
     main_data_out <= vdata_out;
end process;

---  This process block is used to prepare data internally for read or write

  process(main_address,main_data_in,main_select,main_read,main_write,
         pmain_select,pmain_write)

      variable vaccess_mode,vgoe: std_logic;
  begin
-- set default outputs

      nmain_address <= main_address;
      nmain_data_in <= main_data_in;
      nmain_select <= main_select;
      nmain_read <= main_read;
      nmain_write <= main_write;

      vaccess_mode := write_mode;  -- default access mode to write mode
      vgoe := disabled;   -- default global oe to disabled
--   if not selected then we are in read mode
--
  if(pmain_select = '0') then
    vaccess_mode := read_mode;
     vgoe := disabled;  ---   we are not selected turn off global output enable
  elsif(pmain_write = '1') then
     vaccess_mode := write_mode;
     vgoe := disabled;
              -- we are selected but writing, turn off global output enable
  else
     vaccess_mode := read_mode;
     vgoe := enabled;
             -- we are selected and reading, turn on global output enable
  end if;

    global_dataoutput_enable <= vgoe;
    global_write_not_read <= vaccess_mode;

 end process;


---
```

```
---  This is the register block
---
    process(clock,reset,nmain_select,nmain_address,nmain_data_in,nmain_read,
             nmain_write)

      begin

       if(reset = '0') then
         pmain_select <= '0';
         pmain_address <= zero15;
         pmain_data_in <= zero32;
         pmain_read <= '0';
         pmain_write <= '0';
       elsif(clock='1' and clock'event) then
         pmain_select <= nmain_select;
         pmain_address <= nmain_address;
         pmain_data_in <= nmain_data_in;
         pmain_read <= nmain_read;
         pmain_write <= nmain_write;

       end if;
     end process;

     registered_data <= pmain_data_in;
     registered_address <= pmain_address;

end architecture logic;
```

C.2: MISCELLANEOUS MEMORY DECODER

```
--------------------------------------------------------------------------------
--  This block decodes the misc memory space
--------------------------------------------------------------------------------
-- The initialization of the standard library files
--------------------------------------------------------------------------------
library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
--------------------------------------------------------------------------------
entity misc_memory_decoder is

port  (main_address:in std_logic_vector(10 downto 0);
       bad_channel_data,misc_parameters_data:in std_logic_vector(31 downto 0);
       chip_range_data,dedx_data: in std_logic_vector(23 downto 0);
       cluster_data:in std_logic_vector(16 downto 0);
       bad_channel_enable,chip_range_enable,dedx_enable,misc_parameters_enable,
       cluster_enable: out std_logic;
       misc_data_out: out std_logic_vector(31 downto 0)
       );
end entity misc_memory_decoder;

-----------------------------------------------------------------
--Architecture body

architecture logic of misc_memory_decoder  is
--
-- define memory map for address bits 11 downto 8
```

```
--
constant bad_channel_address_space:std_logic_vector(2 downto 0) := "000";
constant chip_range_address_space: std_logic_vector(2 downto 0) := "001";
constant dedx_address_space: std_logic_vector(2 downto 0) := "010";
constant misc_parameters_address_space: std_logic_vector(2 downto 0) := "011";
constant cluster_address_space: std_logic_vector(2 downto 0) := "100";
constant zero8: std_logic_vector(7 downto 0):= "00000000";
constant zero15: std_logic_vector(14 downto 0):= "000000000000000";
constant zero32 : std_logic_vector(31 downto 0):=
"00000000000000000000000000000000";

 begin
--  This process block sets the enables for each address space
process(main_address)
    variable address: std_logic_vector(2 downto 0);
    variable vbc,vcr,vdedx,vmp,vce: std_logic;
   begin
-- set default outputs

   vbc := '0';
   vcr := '0';
   vdedx := '0';
   vmp := '0';
   vce := '0';

 address := main_address(9 downto 7);
             -- only need top 3 bits for address decoding
  case address is
      when bad_channel_address_space =>
         vbc := '1';
      when chip_range_address_space =>
         vcr := '1';
      when dedx_address_space =>
         vdedx := '1';
      when misc_parameters_address_space =>
         vmp := '1';
      when cluster_address_space =>
         vce := '1';
      when others =>
         null;
  end case;

     bad_channel_enable <= vbc;
     chip_range_enable   <= vcr;
     dedx_enable     <= vdedx;
     misc_parameters_enable   <= vmp;
     cluster_enable    <= vce;
  end process;

---  use top three bits of address to mux correct data word out

   with main_address(10 downto 8) select
      misc_data_out <=  bad_channel_data         when "000",
                        zero8&chip_range_data    when "001",
                        zero8&dedx_data          when "010",
                        misc_parameters_data     when "011",
                        zero15&cluster_data      when "100",
                        zero32                   when others;


end architecture logic;
```

C.3: PULSE THRESHOLD CONTROL

```
--------------------------------------------------------------------------------
--  This block decodes the memory addresses for the 8 de/dx parameters
--------------------------------------------------------------------------------
-- Initial Design: Reginald Perry  (07/23/2000)
--------------------------------------------------------------------------------
-- The initialization of the standard library files
--------------------------------------------------------------------------------
library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
--------------------------------------------------------------------------------
entity dedx_threshold_control is

port (control_address: in std_logic_vector(5 downto 0);
                  -- address from the control block
      logic_address  : in std_logic_vector(1 downto 0);
             -- for chiprange design the logic address is always 0
      write_enable, memory_enable   : in std_logic;
             -- read request signal from the control block
      block0,block1,block2,block3: in std_logic_vector(23 downto 0);
                  -- this is the block input data
      read_address : out std_logic_vector(5 downto 0);
                  -- the mux control or logic read_address to the FIFO
      write_not_read0,write_not_read1,write_not_read2,
                  write_not_read3 : out std_logic;
     block_out: out std_logic_vector(23 downto 0)
                           -- this is the block data out
   );
end entity dedx_threshold_control;
--------------------------------------------------------------------
--Architecture body

architecture logic of  dedx_threshold_control is

       signal control_memory_select: std_logic_vector(1 downto 0);
begin

-- let's use a with select to mux read address line
-- use a separate mux for each memory block
-- This is block 0 addresses 0-4
-- need two bits for address
    with memory_enable select
              --- use global_memory_enable to pick read_address
       read_address  <= logic_address when '0',
                        control_address(1 downto 0) when '1',
                        logic_address when others;

-- Define control memory  select value

    control_memory_select <= control_address(3)&control_address(2);

-- use a process block to determine write_not_read lines
process(memory_enable,write_enable,control_memory_select,block0,block1,block2,
        block3)
     variable block_data : std_logic_vector(23 downto 0);
     variable  write_access_required: std_logic;
   begin
```

```
--- use another variable to decide if write access is required
    write_access_required :=  memory_enable and write_enable;

    if(write_access_required = '1') then
               --- need to determine which block wants access

      case control_memory_select is
        when "00" =>
           write_not_read0 <='1';
           write_not_read1 <='0';
           write_not_read2 <='0';
           write_not_read3 <='0';

        when "01" =>
           write_not_read0 <='0';
           write_not_read1 <='1';
           write_not_read2 <='0';
           write_not_read3 <='0';

        when "10" =>
           write_not_read0 <='0';
           write_not_read1 <='0';
           write_not_read2 <='1';
           write_not_read3 <='0';

        when "11" =>
           write_not_read0 <='0';
           write_not_read1 <='0';
           write_not_read2 <='0';
           write_not_read3 <='1';

         when others =>  -- should never come here
           write_not_read0 <='0';
           write_not_read1 <='0';
           write_not_read2 <='0';
           write_not_read3 <='0';
      end case;
    else  --  this a read cyle, set all write_not_read lines low;
           write_not_read0 <='0';
           write_not_read1 <='0';
           write_not_read2 <='0';
           write_not_read3 <='0';
    end if;

--  use another case statement as a mux to set block_data out
     case control_memory_select is
       when "00" =>
         block_data := block0;
       when "01" =>
         block_data := block1;
       when "10" =>
         block_data := block2;
       when "11" =>
         block_data := block3;
       when others =>
         block_data := block0;
      end case;

     block_out <= block_data;

  end process;

end architecture logic;
```

C.4: PULSE MEMORY CONCATENATION

```
--------------------------------------------------------------------------------
--   This block concatinates dedx values into one 36 bit word
--   Include the possiblility for a three bit de/dx but don't do anything with
those values now
--------------------------------------------------------------------------------
-- Initial Design: Reginald Perry   (07/03/2000)
--------------------------------------------------------------------------------
--
--------------------------------------------------------------------------------
library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
--------------------------------------------------------------------------------
entity dedx_memory_concatination is
       port (block0,block1,block2,block3 : in std_logic_vector(23 downto 0);
             dedx_threshold_coarse: out std_logic_vector(35 downto 0);
             dedx_threshold_fine: out std_logic_vector(47 downto 0)
            );
end entity dedx_memory_concatination;

-----------------------------------------------------------------------
--Architecture body
--  memory map
--  dedx_threshold_coarse has coarse level approximations (2 bits)
--  dedx_threshold_fine gives one additional bit between coarse levels
--   Example: note high is (23 downto 12) while low is (11 downto 0)
--         coarse      fine
--   max   FFF
--                     block3_high
--         block1_low
--                     block3_low
--         block0_high
--                     block2_high
--         block0_low
--                     block2_low
--   min   000

architecture logic of dedx_memory_concatination  is
begin
    dedx_threshold_coarse(35 downto 24) <= block1(11 downto 0);
    dedx_threshold_coarse(23 downto 0) <= block0;
    dedx_threshold_fine                <= block3&block2;
end architecture logic;
```

C.5: CLUSTER THRESHOLD CONTROL

```
--------------------------------------------------------------------------------
-- Version 0 This block is used to decide between the read address from the
-- control module or from the logic
--------------------------------------------------------------------------------
-- Initial Design: Shweta Lolage   (07/03/2000)
--------------------------------------------------------------------------------
--
```

```
-- Modified 7/20/2000   Simplify logic  RJP
-- The initialization of the standard library files
--------------------------------------------------------------------------------
library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
--------------------------------------------------------------------------------
entity cluster_threshold_control is
       port (control_address: in unsigned(1 downto 0);
                    -- address from the control block
             logic_address  : in unsigned(1 downto 0);
                    -- for chiprange design the logic address is always 0
             write_enable, memory_enable   : in std_logic;
                    -- read request signal from the control block
             read_address : out unsigned(1 downto 0);
                    -- the mux output read_address to the FIFO
             write_not_read : out std_logic
            );
end entity cluster_threshold_control;

-----------------------------------------------------------------------
--Architecture body

architecture logic of  cluster_threshold_control is
begin

-- let's use a with select to mux read address line
-- we only have 3 types so set top three bits of address space equal to zero
    with memory_enable select   --- use test_enable to pick read_address
       read_address  <= logic_address when '0',
                        control_address when '1',
                        logic_address when others;

--   use simple data_path statement for write_not_read
---  write_not_read is write_enable and test_enable

    write_not_read <= memory_enable and write_enable;


end architecture logic;
```

C.6: CHIP RANGE CONTROL

```
--------------------------------------------------------------------------------
-- Version 0 This block is used to decide between the read address from the
-- control module or from the logic
--------------------------------------------------------------------------------
-- Initial Design: Shweta Lolage   (07/03/2000)
--------------------------------------------------------------------------------
--
-- Modified 7/20/2000   Simplify logic  RJP
-- The initialization of the standard library files
--------------------------------------------------------------------------------
library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
```

```
--------------------------------------------------------------------------------
entity chiprange_control is
       port (control_address: in unsigned(5 downto 0);
              -- address from the control block
             write_enable, memory_enable   : in std_logic;
                     -- read request signal from the control block
             read_address : out unsigned(1 downto 0);
                     -- the mux output read_address to the FIFO
             write_not_read : out std_logic
            );
end entity chiprange_control;

-------------------------------------------------------------------
--Architecture body

architecture logic of chiprange_control  is

       constant logic_address: unsigned(5 downto 0) := "000000";
begin


-- let's use a with select to mux read address line

    with memory_enable select   --- use test_enable to pick read_address
       read_address  <= logic_address(1 downto 0) when '0',
                        control_address(1 downto 0) when '1',
                        logic_address(1 downto 0) when others;

--   use simple data_path statement for write_not_read
---  write_not_read is write_enable and test_enable

    write_not_read <= memory_enable and write_enable;


end architecture logic;
```

C.7: MISCELLANEOUS DOWNLOADED PARAMETER SPLITTER

```
--------------------------------------------------------------------------------
--   This block splits the 32 bit long misc downloaded parameter word into
--   individual blocks
--------------------------------------------------------------------------------
-- Initial Design: Reginald Perry   (07/03/2000)
--------------------------------------------------------------------------------
--
--------------------------------------------------------------------------------
library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
--------------------------------------------------------------------------------
entity misc_downloaded_parameter_splitter is
       port (data_in : in std_logic_vector(31 downto 0);
             dp_test_bit,dp_disable_bit: out std_logic;
             dp_smt_id: out std_logic_vector(2 downto 0);
             dp_delay_count,dp_seq_id: out std_logic_vector(7 downto 0);
             dp_hdi_id: out std_logic_vector(2 downto 0)
            );
```

```
end entity misc_downloaded_parameter_splitter;

-------------------------------------------------------------------
--Architecture body

architecture logic of misc_downloaded_parameter_splitter  is

begin

   dp_test_bit        <= data_in(28);
   dp_disable_bit     <= data_in(27);
   dp_smt_id          <= data_in(26 downto 24);
   dp_delay_count     <= data_in(23 downto 16);
   dp_seq_id          <= data_in(15 downto 8);
   dp_hdi_id          <= data_in(2 downto 0);

end architecture logic;
```

C.8: TEST DATA CONTROL

```
-------------------------------------------------------------------------------
-- Version 0 This block is used to decide between the read address from the
-- control module or from the logic
-------------------------------------------------------------------------------
-- Initial Design: Shweta Lolage   (07/03/2000)
-------------------------------------------------------------------------------
--
-- Modified 7/20/2000   Simplify logic  RJP
-- The initialization of the standard library files
-------------------------------------------------------------------------------
library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
-------------------------------------------------------------------------------
entity testdata_control is
       port (control_address: in unsigned(7 downto 0);
                -- address from the control block
             logic_address  : in unsigned(7 downto 0);
                -- address form the logic for the bad channels
             write_enable, test_enable   : in std_logic;
                           -- read request signal from the control block
             read_address : out unsigned(7 downto 0);
                -- the mux output read_address to the FIFO
             write_not_read : out std_logic
             );
end entity testdata_control;

-------------------------------------------------------------------
--Architecture body

architecture logic of testdata_control  is

       constant zero11 : unsigned (10 downto 0):= "00000000000";

begin

-- let's use a with select to mux read address line
```

```
    with test_enable select   --- use test_enable to pick read_address
       read_address  <= logic_address when '0',
                        control_address when '1',
                        logic_address when others;

--   use simple data_path statement for write_not_read
---  write_not_read is write_enable and test_enable

    write_not_read <= test_enable and write_enable;


end architecture logic;
```

C.9: BAD CHANNEL CONTROL

```
-------------------------------------------------------------------------------
-- Version 0 This block is used to decide between the read address from the
-- control module or from the logic
-------------------------------------------------------------------------------
-- Initial Design: Shweta Lolage   (07/03/2000)
-------------------------------------------------------------------------------
-- Modified 7/20/2000   Simplify logic  RJP
-- The initialization of the standard library files
-------------------------------------------------------------------------------
library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
-------------------------------------------------------------------------------
entity badchannel_control is
       port (control_address: in unsigned(10 downto 0);
                     -- address from the control block
             logic_address  : in unsigned(10 downto 0);
                     -- address form the logic for the bad channels
             write_enable, memory_enable   : in std_logic;
                     -- read request signal from the control block
             read_address : out unsigned(10 downto 0);
                     -- the mux output read_address to the FIFO
             write_not_read : out std_logic
          );
end entity badchannel_control;
-------------------------------------------------------------------
--Architecture body
architecture logic of badchannel_control  is

begin

-- let's use a with select to mux read address line

    with memory_enable select   --- use test_enable to pick read_address
       read_address  <= logic_address when '0',
                        control_address when '1',
                        logic_address when others;
--   use simple data_path statement for write_not_read
---  write_not_read is write_enable and test_enable

    write_not_read <= memory_enable and write_enable;

end architecture logic;
```

C.10: BAD CHANNEL MULTIPLEXER

```
--------------------------------------------------------------------------------
-- Version 1 This code decodes the data for the bad channels as the data
-- from the control is down loaded in set of 32 words, with each word
representing
-- the status of 32 channels. This logic finds out the individual channel
status.
--------------------------------------------------------------------------------
-- Initial Design: Shweta Lolage   (6/27/2000)
-- Simplify Logic: RJP (7/21/2000)
--------------------------------------------------------------------------------
-- Initializations to the standard library files
--------------------------------------------------------------------------------
library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
--------------------------------------------------------------------------------
entity bad_channel_mux is

port  (data_word      : in unsigned (31 downto 0);
       address        : in integer range 31 downto 0;
       bad_channel_status : out std_logic
       );
end entity bad_channel_mux;

--------------------------------------------------------------------------------
---
--Architectural body of the design

architecture logic of bad_channel_mux is

begin

-- use with select to mux

    with address select
           bad_channel_status <= data_word(0) when 0,
                                 data_word(1) when 1,
                                 data_word(2) when 2,
                                 data_word(3) when 3,
                                 data_word(4) when 4,
                                 data_word(5) when 5,
                                 data_word(6) when 6,
                                 data_word(7) when 7,
                                 data_word(8) when 8,
                                 data_word(9) when 9,
                                 data_word(10) when 10,
                                 data_word(11) when 11,
                                 data_word(12) when 12,
                                 data_word(13) when 13,
                                 data_word(14) when 14,
                                 data_word(15) when 15,
                                 data_word(16) when 16,
                                 data_word(17) when 17,
                                 data_word(18) when 18,
                                 data_word(19) when 19,
                                 data_word(20) when 20,
                                 data_word(21) when 21,
```

```
                                data_word(22) when 22,
                                data_word(23) when 23,
                                data_word(24) when 24,
                                data_word(25) when 25,
                                data_word(26) when 26,
                                data_word(27) when 27,
                                data_word(28) when 28,
                                data_word(29) when 29,
                                data_word(30) when 30,
                                data_word(31) when 31,
                                data_word(0) when others;

end architecture logic;
```

C.11: GAIN OFFSET CONTROL

```
--------------------------------------------------------------------------------
-- Version 0 This block is used to decide between the read address from the
-- control module or from the logic
--------------------------------------------------------------------------------
-- Initial Design: Shweta Lolage   (07/03/2000)
--------------------------------------------------------------------------------
--
-- Modified 7/20/2000   Simplify logic  RJP
-- The initialization of the standard library files
--------------------------------------------------------------------------------
library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
--------------------------------------------------------------------------------
entity gainoffset_control is

       port (control_address: in unsigned(11 downto 0);
                    -- address from the control block
             logic_address  : in unsigned(11 downto 0);
                     -- address form the logic for the bad channels
             write_enable, memory_enable   : in std_logic;
                     -- read request signal from the control block
             read_address : out unsigned(11 downto 0);
                    -- the mux output read_address to the FIFO
             write_not_read : out std_logic
            );
end entity gainoffset_control;

--------------------------------------------------------------------
--Architecture body

architecture logic of gainoffset_control  is

begin

-- let's use a with select to mux read address line

    with memory_enable select   --- use test_enable to pick read_address
       read_address  <= logic_address when '0',
```

```
                        control_address when '1',
                        logic_address when others;

--   use simple data_path statement for write_not_read
---  write_not_read is write_enable and test_enable

    write_not_read <= memory_enable and write_enable;


end architecture logic;
```

C.11: SMT DATA FILTER

```
--------------------------------------------------------------------------------
-- Revision 2.1 SMT Data Filter
-- Based on 5/26/2000 Document -- Earle, Specifications for One Channel of the
STC Logic
--------------------------------------------------------------------------------
-- Initial Design: Reginald J. Perry   (6/4/2000)
--------------------------------------------------------------------------------
--  Modified 6/22/00 based on new specs:  Reginald J. Perry
--     Add 18 bit output (double byte wide)
--  Modified 7/14/00  RJP
--    Add delay counter for event number
-- Modified 8/02/00 Shweta Lolage
       -- to change the format of the smt_out to fit the specifications
       -- and minor changes to filter the excess C0's
-- Modified 10/05/00 Shweta Lolage
-- to remove the test inputs as they will be mux-ed with the FIFO output
-- to go directly to the Strip reader control
--------------------------------------------------------------------------------
-- This block filters raw SMT data before it is stored in the input FIFO
-- Complex delay counter currently does not fit.  Implement simple delay
counter for now
--
--
library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
--------------------------------------------------------------------------------
entity smt_data_filter is
       port (delay_in: in unsigned(7 downto 0);
                 -- this is the delay in FRC start signal
             frc_start: in std_logic;
                    -- this indicates that FRC data has arrived
             disable: in std_logic;
                      --- this bit is used to turn off the channel
             event_in: in unsigned(7 downto 0); -- this is the event number
             reset,clock,cav,dav,lnkrdy,smt_error_in: in std_logic;
             smt_in: in unsigned(7 downto 0);
             end_of_event: out std_logic;  -- specifies the end of event
             smt_out: out unsigned(17 downto 0);
             wrreq: out std_logic
            );
end entity smt_data_filter;

----------------------------Architecture -------------------------
```

```
architecture logic_design of smt_data_filter is

---------------------Signal and constant Declaration-------------------------
       signal delay_eq_zero,tdelay: std_logic;
       signal nserr,pserr: std_logic;  --  this bit is used to record the serr
       signal nsmt_out,psmt_out: unsigned(17 downto 0);
              -- this register is used to hold smt data
       signal ndelay,pdelay: unsigned(7 downto 0);
              -- this is the internal delay counter
       constant zero18: unsigned(17 downto 0) := "000000000000000000";
       constant zero8: unsigned(7 downto 0) := "00000000";
       constant zero_address: unsigned(7 downto 0) := "00000000";
       constant zC0: unsigned(7 downto 0) := "11000000";
       type mystates is (s0,s1,s2,s3,s4);
       signal ns,ps,ndcs,pdcs: mystates;

begin

--  need two states to write data
process(ps, psmt_out,smt_error_in,cav,dav,lnkrdy,event_in,pdelay,pserr)
         variable vsmt_enabled: std_logic;
         variable vsmt_in,vupper,vlower: unsigned(7 downto 0);
 begin

--   set default outputs
     ns <= ps;
     nserr <= pserr;
     end_of_event <= '0';
     nsmt_out <= psmt_out;
     vupper := psmt_out(15 downto 8);   --  this is the last data byte
     vsmt_in := smt_in;  -- use variable to hold the input stream data.
                                         -- this is the real data

---  SMT data is valid if
--- dav = lnkrdy=0 and cav=1 and enable = 1 and disable = 0
---  use a simple logic expression for this.  Save in a variable
      vsmt_enabled := (not dav) and (not lnkrdy) and cav  and (not disable);

--  if channel is active then access the state machine
     if (vsmt_enabled= '1') then

       case ps is
        when s0 =>
               --- this state is also used to initialize the delay counter.  See
       process delay_counter
          end_of_event <= '0';
          wrreq <= '0';
          nsmt_out(17) <= smt_error_in;
               nsmt_out(15 downto 8) <= vsmt_in;
                    -- this reads the upper byte of the word
--
--  we were told that you can't start up with C0s.
-- So no need to filter them here
--  go immediately to s1
--
          ns <= s1;

       when s1 =>
--   this is the lower byte
--   If upper byte is C0 then go to s2 to wait for delay counter to be zero to
write event number
--   into trailer
          end_of_event <= '0';
```

```
          if(vupper = zC0) then
            ns <= s2;
            wrreq <= '0';
          else
--    we have not seen trailer C0 in upper byte, use smt_in for lower byte.
            wrreq <= '0';   --- we will enable output FIFO in next state
                            --- use this state to load register
            vlower := vsmt_in;
            end_of_event <= '0';   -- we are not at end of event
            ns <= s3;
              --  use s3 to write current word to FIFO and read next upper byte
            nsmt_out(7 downto 0) <= vlower;
                    nsmt_out(16) <= smt_error_in;

         end if;
--  this is the end of the current channel,
-- must wait for delay_eq_zero to assert before writing trailer
        when s2 =>
           end_of_event <= '0';
              wrreq <= '0';
                      ---  need to wait in here for delay to go to zero
           nsmt_out <= '1'&pserr&zC0&event_in;
              -- need to use dummy state to actually write FIFO.  See s3 below

           if(delay_eq_zero = '1') then
             -- write out the trailer if delay is zero
             -- for the trailer, use msb for eof and use the next bit for serr
             ns <= s4;   --  go back to s0 and start over
           else
             ns <= s2;
           end if;

         when s3 =>
             -- use this state to write last word and read next upper byte
             -- for non end of event
          end_of_event <= '0';
             --  set serr based on two error bits and current value of serr. --
             -- Use boolean equation
          nserr <= psmt_out(17) or psmt_out(8) or pserr;
          wrreq <= '1';
               -- wrreq is not latched so it goes high immediately when we enter
               -- s3.  Data word will be latched on the next clock edge. Timing
               -- is a bit problematic here because new
               -- dataword is being written at the same time.  Therefore, --
               -- output FIFO must latch on
                --  rising edge of clock when wrreq is enabled.

          nsmt_out(17) <= smt_error_in;
               nsmt_out(15 downto 8) <= vsmt_in;
                     -- this reads the next upper byte of the word
          ns <= s1;  -- go to s1 to see if this is a C0

        when s4 =>   -- this is the write for an end of event
         end_of_event <= '1';
         wrreq <= '1';
              if( vsmt_in = zC0) then
-- if the next byte after the end of event is still C0 then wait in this state
-- till you get anything other than C0.
                    ns <= s4;
              else
               ns <= s0;  -- go to s0 to reset delay counter
              end if;
```

```
       when others =>   -- should never come here
         ns <= s0;
     end case;
     else
--  if channel goes off or we are disabled, reset to s0
       wrreq <= '0';
       end_of_event <= '0';
       ns <= s0;
     end if;
end process;

--   This process block implements the simple "delay" counter.  Not exactly
sure what it is used for RJP
delay_counter: process(pdelay, ps, delay_in,frc_start,pdcs)
                variable temp_delay: std_logic;
                variable i: integer;
                begin
                  ndelay <= pdelay;
                  ndcs <= pdcs;

             --  use a variable to determine if current count is zero
                   temp_delay := '0';
                   for i in 7 downto 0 loop
                     temp_delay := temp_delay or pdelay(i);
                    -- OR all bits to determine if we are at zero
                   end loop;

                   tdelay <= temp_delay;

-- this is the main state machine for delay counter

                  case pdcs is
                   when s0 =>
                    ---   we are waiting to start the counter in this state
                     delay_eq_zero <= '0';
                     ndelay <= delay_in;
                     ndcs <= s0;  --  come back here if channel is inactive
                     if(ps=s1) then
                       ndcs <= s1;     --  smt data has started counter
                     end if;
                     if(frc_start = '1') then
                       ndcs <= s4;     --- frc_start has started counter
                     end if;

--  For simple delay counter, we will only have three states.
                   when s1 =>
                     delay_eq_zero <= '0';
                    if(tdelay = '0')  then
                      ndcs <= s3;   -- go to s3 to set delay_eq_zero = 1
                     else
                      ndelay <= pdelay - 1;    -- decrement ndelay
                      ndcs <= s1;  -- come back to this state
                    end if;
                    if(frc_start = '1') then   --- should be able to combine
with above but synth tool choked on this
                      ndcs <= s3;
                    end if;
--  in this state we set delay_eq_zero = 1 and hold it until main FSM resets by
-- going back to s0
                   when s3 =>
                     delay_eq_zero <= '1';
-- check if main FSM is in s0, if so reset this FSM to S0 otherwise stay here
                     if(ps = s0) then
```

```
                     ndcs <= s0;
                    else
                     ndcs <= s3;
                    end if;
--
-- use smt data to stop counter
--
                  when s4 =>
                    delay_eq_zero <= '0';
                   if(tdelay = '0')  then
                     ndcs <= s3;   -- go to s3 to set delay_eq_zero = 1
                    else
                     ndelay <= pdelay - 1;    -- decrement ndelay
                     ndcs <= s4;  -- come back to this state
                   end if;
                   if(ps=s1) then   --- should be able to combine with above
but synth tool choked on this
                     ndcs <= s3;
                   end if;

                  when others =>   -- should never come here
                     ndcs <= s0;
                    delay_eq_zero <= '0';
                 end case;
end process;

--  register block
reg: process(ns,ndcs,nsmt_out,clock,reset,ndelay,nserr)
 begin
  if(reset = '0') then
      ps <= s0;
      pdcs <= s0;
      psmt_out <= zero18;
      pdelay <= zero8;
      pserr <= '0';
  elsif (clock'event and clock='1') then
      ps <= ns;
      pdcs <= ndcs;
      psmt_out <= nsmt_out;
      pdelay <= ndelay;
      pserr <= nserr;
  end if;
end process reg;


 smt_out <= psmt_out; -- this goes to the output fifo

end architecture logic_design;   --End of architecture block
```

C.12: SMT TEST SELECT

```
--------------------------------------------------------------------------------
-- Version 0 This block is used to decide between the read address from the
-- control module or from the logic
--------------------------------------------------------------------------------
-- Initial Design: Shweta Lolage   (10/05/2000)
--------------------------------------------------------------------------------
-- The initialization of the standard library files
--------------------------------------------------------------------------------
```

```
library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
--------------------------------------------------------------------------------
entity smt_test_select is
port  (clock,reset : in std_logic;
       test_data, fifo_data : in unsigned (17 downto 0);
       l3_raw_data_eof : in std_logic;
              -- this signal is taken from the strip reader control
       event_start, test , read_data: in std_logic;
              -- event start is the main signal which indicates start of an --
              -- event for the channel
              -- test input this indicates whether the strip reader is going to
              -- process real data
              -- or this event is a test run
              -- read data is the signal from the strip reader
              -- to read the next 18 bit word
       fifo_empty : in std_logic; -- signal from the smt fifo;
       logic_address  : out unsigned(7 downto 0); -- for test lut
       strip_reader_data : out unsigned (17 downto 0);
       read_fifo, memory_fifo_empty : out std_logic
              -- read_req signal to the FIFO
              -- fifo or memory empty signal for strip reader
    );
end entity smt_test_select;

-----------------------------------------------------------------
--Architecture body

architecture logic of smt_test_select  is

type mystates is (stest,sstart,sfirst,saddress);

signal ps, ns : mystates;
signal naddress, paddress : unsigned (7 downto 0);
signal read_test : std_logic;
       -- signal indicating to increment the address for next word
signal int_test : std_logic;
       -- this signal is used to latch the test signa
       -- I am not sure whether this signal
       -- will remain high all the time test data is running
       -- or will be a pulse indicating test mode
signal test_lut_empty : std_logic;

constant zero8 : unsigned(7 downto 0) := "00000000";
constant zero18 : unsigned (17 downto 0) := "000000000000000000";
constant max_count : unsigned (7 downto 0) := "11111111";

begin


main: process (ps,test,event_start,paddress)

       begin

       naddress <= paddress;

              case ps is

                     when stest =>
```

```
                        int_test <= '0';
-- check for the test input to go to the next state
                        if (test = '1') then
                              int_test <= '1';
                              ns <= sstart;

                        else
                              int_test <= '0';
                              ns <= stest;
                        end if;

                  when sstart =>
-- initialize the adddress and the testlut empty signal
                        naddress <= zero8;
                        test_lut_empty <= '0';

                        if(event_start = '1') then
                              ns <= sfirst;
                        else
                              ns <= sstart;
                        end if;


                  when sfirst =>
                        -- this the word read out of the test lut
                        if(read_test = '1') then
                              naddress <= zero8; -- this just to make sure
that the first word is read out
                              ns <= saddress;
                        else
                              ns <= sfirst;
                        end if;

                  when saddress =>
      -- the machine will remain in this state till the reset signal comes in
                        if(read_test = '1') then
                           naddress <= paddress + 1;
                        else
                           naddress <= paddress;
                        end if;

                        if (l3_raw_data_eof = '0') then
                           ns <= saddress;
                        else
                           ns <= stest;
                        end if;

                  when others =>
                        ns <= stest;

            end case;

      end process main;


process(int_test,read_data)

variable vread_test, vread_fifo : std_logic;

      begin

            if (ps = sfirst or ps = saddress) then
```

```
                       vread_test := read_data;
                       vread_fifo := '0';
                else

                       vread_test:= '0';
                       vread_fifo:= read_data;
                end if;

                read_test <= vread_test;
                read_fifo <= vread_fifo;

end process;

reg: process (clock,reset,ps, paddress)

                begin

                if (reset = '0') then
                       ps <= stest;
                       paddress <= zero8;

                elsif( clock'event and clock ='1') then
                       ps <= ns;
                       paddress <= naddress;
                end if;

       end process;


-- with select to mux read data

    with int_test select   --- use test_enable to pick read_address
       strip_reader_data  <= fifo_data when '0',
                             test_data when '1',
                             zero18 when others;

    with int_test select   --- use test_enable to pick read_address
       memory_fifo_empty  <= fifo_empty when '0',
                             test_lut_empty when '1', -- this just a dummy
signal
                             '0' when others;


logic_address <= paddress;

end architecture logic;
```

C.13: STRIP READER CONTROL

```
--------------------------------------------------------------------------------
-- Revision 2.1 Strip Reader Control
-- Based on 6-22-00 Document
--------------------------------------------------------------------------------
-- Modified by Shweta Lolage(04-10-2000)
-- Modified by Reginald Perry (6-4-2000)
--     Add comments and fix up notation
--     Add some monitoring counters
-- Modified by Reginald Perry (6-5-2000)
--     Add additional monitoring counters
-- Modified by Reginald Perry (6-22-2000)
```

```
--  Add 18 bit SMT input
-- Modified by Reginald Perry (7-12-2000)
--  Add monitor clear input
--  Add chip monitor counters
--- Modified by RJP (7/21/2000)
--  Need to add rawdata l3 output
-- Modified to add the initial state by Shweta Lolage (09-07-2000)
--------------------------------------------------------------------------------
library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library cluster_package;
use cluster_package.cluster_package.all;

--------------------------------------------------------------------------------
--    This block defines the inputs and outputs of the counter              -
--------------------------------------------------------------------------------
entity strip_reader_control is
       port(reset,clock,input_fifo_empty,output_fifo_full,monitor_clear,
            event_start : in  std_logic;
            smt_in: in unsigned(2 downto 0);
            bad_channel_data :in std_logic;
                    -- bad channel data is read from a rom
            stream_in: in unsigned(17 downto 0);
                     -- MSB of stream_in is Error bit from SMT channel
            seqid: in unsigned(7 downto 0);
            hdiid: in unsigned(2 downto 0);
            corrected_data:in unsigned (7 downto 0);
                    ---  this is input data_type parameteric data in the format
                    --- of axial, stereo, and ninety_degrees
                    --- (low followed by high)
            data_range_in: in unsigned(23 downto 0);--  start of output data
            oread,owrite:out std_logic;
                    --  signals used to enable input and output fifo
            ostate :out unsigned(3 downto 0);
            bad_channel_address:out unsigned(10 downto 0);
            chip_data : out unsigned (11 downto 0);
            output:out unsigned (22 downto 0);
                    --  monitoring data
            l3_correcteddata: out unsigned(24 downto 0);
                    -- note l3_out(22) is the end of event bit
            chip_monitor: out chip_array;
            smt_monitor: out smt_array;
            l3_rawdata: out unsigned(17 downto 0);
            l3_rawdata_eof,l3_rawdata_enable: out std_logic
           );
end entity strip_reader_control;

----------------------------Architecture Begining--------------------------

architecture logic_design of strip_reader_control is

---------------------Signal and constant Declaration-------------------------

signal neof,peof,nmismatchbit,pmismatchbit,pbc,nbc,nserr,pserr: std_logic;
signal pnew_event,nnew_event: std_logic;
signal n_new_data_flag,p_new_data_flag: std_logic;
signal nseq_id,pseq_id: unsigned(7 downto 0);
signal nhdi_id,phdi_id: unsigned(2 downto 0);
signal noutput,poutput    : unsigned (22 downto 0);
signal nchip_id,pchip_id  : unsigned (3 downto 0);
```

```
signal nchan_id,pchan_id  : unsigned (6 downto 0) ;
signal ndata_type, pdata_type: unsigned(1 downto 0);
signal nstate, pstate   : unsigned(3 downto 0);
signal nbad_channel_address, pbad_channel_address: unsigned(10 downto 0);
signal nchip_data,pchip_data : unsigned (11 downto 0);
-----
---  These signals are for monitoring information
---
signal nsmt_mismatch_count,psmt_mismatch_count,nsmt_error_count,
       psmt_error_count: unsigned(23 downto 0);
signal nsmt_zero_error,psmt_zero_error,nl3_trailer,
       pl3_trailer: unsigned(23 downto 0);

--  These signals are used in the clear operation for the monitor counters.
signal nchip,pchip: chip_array;  -- chip array is defined in cluster_package

constant zero9  : unsigned (8 downto 0)  := "000000000";
constant zero8  : unsigned (7 downto 0)  := "00000000";
constant zero7  : unsigned (6 downto 0)  := "0000000";
constant zero4  : unsigned (3 downto 0)  := "0000";
constant zero3  : unsigned (2 downto 0)  := "000";
constant C0     : unsigned (7 downto 0)  := "11000000";
constant zero2  : unsigned (1 downto 0)  := "00";

constant initial_state            : unsigned (3 downto 0):= "0000";
constant first_read_state   : unsigned (3 downto 0):= "0001";
constant seq_hdi_state      : unsigned (3 downto 0):= "0010";
constant chip_zero_state    : unsigned (3 downto 0):= "0011";-- State constants
constant chan_value_state   : unsigned (3 downto 0):= "0100";
constant wait_for_data_state : unsigned (3 downto 0):= "0101";
constant correctvalue_state  : unsigned (3 downto 0):= "0110";
constant write_fifo_state    : unsigned (3 downto 0):= "0111";
constant read_fifo_state     : unsigned (3 downto 0):= "1000";
constant read_wait_state    : unsigned (3 downto 0):= "1001";

constant zero23 : unsigned (22 downto 0) := "00000000000000000000000";
constant zero24 : unsigned (23 downto 0) := "000000000000000000000000";

constant high   : std_logic := '1';
constant low    : std_logic := '0';

-- constant
constant chip0 : unsigned(3 downto 0) := "0000";
constant chip1 : unsigned(3 downto 0) := "0001";
constant chip2 : unsigned(3 downto 0) := "0010";
constant chip3 : unsigned(3 downto 0) := "0011";
constant chip4 : unsigned(3 downto 0) := "0100";
constant chip5 : unsigned(3 downto 0) := "0101";
constant chip6 : unsigned(3 downto 0) := "0110";
constant chip7 : unsigned(3 downto 0) := "0111";
constant chip8 : unsigned(3 downto 0) := "1000";
constant chip9 : unsigned(3 downto 0) := "1001";

--  valid data types are now defined in cluster package
--constant undefined_type: unsigned(1 downto 0) := "00";
--constant stereo_type : unsigned(1 downto 0)  := "01";
--constant axial_type: unsigned(1 downto 0) := "10";
--constant ninety_degrees_type: unsigned(1 downto 0) := "11";


-- to define the states of the decoder --
type mystates is
(sinit,sfirst_word,seq_hdi,chip_zero,chan_value,wait_for_data,fill_output_fifo,
```

```
 write_fifo,read_fifo,read_wait);
signal ns,ps,nrs,prs : mystates;
type event_states is (sinit,snew_event,swait1,swait2,sevent);
signal nevent_state,pevent_state: event_states;

-----------------------------Block Begins-----------------

begin

process(pstate,ps,prs,poutput,pchip_id,pchan_id,p_new_data_flag,pdata_type,
        corrected_data,peof,pseq_id,phdi_id,bad_channel_data,seqid,hdiid,
       input_fifo_empty,psmt_mismatch_count,psmt_zero_error,pmismatchbit,
       output_fifo_full,pchip,monitor_clear,pbc,pserr,pnew_event)

---  Variable declaration ---
---  Use variables to much of the state information.  Need to include a
definition table (RJP 6/4/00)

 variable voutput : unsigned ( 22 downto 0);
 variable vnew_data_flag,vend_of_event: std_logic;
 variable verror_high,verror_low: unsigned(0 downto 0);
 variable vchip_id : unsigned(3 downto 0);
 variable vchan_id: unsigned(6 downto 0);
 variable vstream_high_byte,vstream_low_byte,vseq_id: unsigned(7 downto 0);
 variable vhdi_id: unsigned(2 downto 0);
 variable vdata_type:unsigned(1 downto 0);
 variable next_state,return_state: mystates;
 variable vstate : unsigned(3 downto 0);
 variable veof,vbc,vserr: std_logic;


     begin
--
--  Default Outputs
--

--
--  Default variable values
--
    vstate := pstate;
    vnew_data_flag := p_new_data_flag;
    next_state := ps;
    voutput := poutput;
    veof := peof;
    vdata_type := pdata_type;
    vchip_id := pchip_id;
    vchan_id := pchan_id;
    vseq_id := pseq_id;
    vhdi_id := phdi_id;
    vbc := pbc;
    vserr := pserr;
    verror_high(0) := stream_in(17);
    verror_low(0)  := stream_in(16);
    vstream_high_byte := stream_in(15 downto 8); -- MSB of stream data
    vstream_low_byte := stream_in(7 downto 0);  -- LSB of stream data
    return_state := prs;
    nchip_data <= pchip_data;

---  smt counters
    nsmt_mismatch_count <= psmt_mismatch_count;
    nsmt_zero_error <= psmt_zero_error;
    nmismatchbit <= pmismatchbit;
    nbad_channel_address <= pchip_id&pchan_id;
```

```
            --  this is used to address bad channel prom

--
--  Chip counters -- default outputs
--
    nchip <= pchip;
--
--  Check for serr
--
    vserr := verror_high(0) or verror_low(0) or pserr;

--  set l3_rawdata to defaults.
    l3_rawdata <= stream_in;
    l3_rawdata_eof <= '0';
    l3_rawdata_enable <= '0';

---------------------
----CASE STATEMENT --
---------------------
-- The following case statment will look at the incoming stream of data and --
will proceed to
-- compare and check for the validity of the same.  For each case different
actions will occur.
-- Note1: Every state requires to have a read fifo before it can proceed to
check the information.
-- Note2: After the channel state and 11-bit word composed of the chip and
channel id are sent to
-- to an lpm rom, which will send an 9 bit word back into the decoder.  From
this 9 bit word we
-- will look at the MSB and if it is "0" then we know that the channel is ok.
-- Note3: Only the channel and Value state will have the command to enable
writing to the FIFO.

   CASE ps IS

       -- the initial state in which the FSM is waiting for the event start
             When sinit =>
                   vstate := initial_state;
                   if (event_start = '1') then
                         next_state := sfirst_word;
                   else
                         next_state := sinit;
                   end if;
       -- the initial state in which the first word is read outof the fifo
             when sfirst_word =>
                   vstate := first_read_state;
                   if (pevent_state = sevent) then
                         next_state := read_fifo;
                         return_state := seq_hdi;
                   else
                         next_state := sfirst_word;
                   end if;


          -- Sequence State
             -- Check for correct sequence ID and hdi_id  --
        WHEN seq_hdi =>
                         vstate := seq_hdi_state;
  -- variable used to check current state this is for debugging only
               next_state := read_fifo;
               return_state := chip_zero;
               vnew_data_flag := high;
                     -- set new data high when looking for seqid and hdiid
```

```
                vseq_id := vstream_high_byte;    --  save the sequence id
                vhdi_id := vstream_low_byte(2 downto 0);
                          --- save the hdi id.  It is bits 2,1,0
                nmismatchbit <= low;   -- reset mismatch bit
       if((vstream_high_byte /= seqid) or (vstream_low_byte /= hdiid)) then
                    --when stream does not equal the preset value

--  need to count the number of errors in seq_Id mismatch
                   nsmt_mismatch_count <= psmt_mismatch_count + 1;
                   nmismatchbit <= high;   -- set the mismatch bit high
                end if;
 --  need to enable l3_rawdata so that l3 will latch when we change states
                l3_rawdata_eof <= '0';
                l3_rawdata_enable <= '1';

             -- CHIP State
         -- Check for correct chip ID and zero_byte
           WHEN chip_zero =>
                vstate := chip_zero_state;
                next_state := read_fifo;
                if(vstream_low_byte /= zero7) then
                          -- check that low byte is zero
                  nsmt_zero_error <= psmt_zero_error + 1;
                end if;
                if (vstream_high_byte(7) = high) then
                           ---  if MSB = 1 this is a new chip.  Save chip ID
                   vchip_id    := vstream_high_byte(3 downto 0);
                          --  and return to channel value
                   return_state := chan_value;
                else
                   return_state := chip_zero;
                    ---  If MSB is not 1 then error come back to this state
                end if;                      ---  and look for valid chip id
       --  need to save l3 raw
                l3_rawdata_eof <= '0';
                l3_rawdata_enable <= '1';

             -- CHANNEL State --
          -- Get Channel ID and Value from Stream
             WHEN chan_value =>            -- The channel ID is read from stream
                          vstate := chan_value_state; --
                next_state := read_fifo;
                return_state := chan_value;
                veof := '0';
                if (vstream_high_byte = C0) then
                    -- if channel is C0 then we are at the end of event
                   veof     := high;
                          -- event number is in low_byte so save it in voutput
                   voutput(18 downto 11) :=  vstream_low_byte;
                   next_state := fill_output_fifo;
                           --  jump to fill_output_fifo
                             return_state := sinit;
                                 -- return to next sequence ID after write fifo
                   l3_rawdata_enable <= '1';
                   l3_rawdata_eof <= '1';      -- set end of event bit
                 elsif (vstream_high_byte(7) = '1') then
                          -- This means this is a new chip ID
                   next_state := chip_zero;
                          --  Check for a new type of chip. Do NOT read fifo.
                   l3_rawdata_enable <= '0';      --  do not write into l3
                   l3_rawdata_eof <= '0';
                 else
                   next_state := wait_for_data;
```

```
                          --  this is a regular channel/data pair
                    vchan_id := vstream_high_byte(6 downto 0);
                             -- need to goto a wait state to give LUT enough
                    nchip_data <= pchip_id&vstream_low_byte;
                            -- time to access corrected data
                    l3_rawdata_eof <= '0';
                    l3_rawdata_enable <= '1';   -- enable l3 rawdata
                   end if;                          -- come back here after write

              -- wait for data State --
  -- This is a dummy state to wait for the corrected data to come from the LUT.
               WHEN wait_for_data =>
                    vstate := wait_for_data_state;
                    next_state := fill_output_fifo;
                            -- go to fill output fifo state
---
---   Data is valid, Increment chip monitor counters here
--
--  Chip counters -- default outputs
--
                  nchip <= pchip;
                  case vchip_id is
                     when chip0 =>
                         nchip(0) <= add_one(pchip(0));
                     when chip1 =>
                         nchip(1) <= add_one(pchip(1));
                     when chip2 =>
                         nchip(2) <= add_one(pchip(2));
                     when chip3 =>
                         nchip(3) <= add_one(pchip(3));
                     when chip4 =>
                         nchip(4) <= add_one(pchip(4));
                     when chip5 =>
                         nchip(5) <= add_one(pchip(5));
                     when chip6 =>
                         nchip(6) <= add_one(pchip(6));
                     when chip7 =>
                         nchip(7) <= add_one(pchip(7));
                     when chip8 =>
                         nchip(8) <= add_one(pchip(8));
                     when others =>
                         nchip <= pchip;   --- should never come here;
                    end case;

 --  DO NOT enable l3 raw here
                l3_rawdata_eof <= '0';
                l3_rawdata_enable <= '0';

                    WHEN fill_output_fifo =>

                      vstate := correctvalue_state;
--  This is a bit backward but if veof leave voutput(18 downto 11) only.  This
was set in chan_value state
--  if not end of event that check for a bad channel and replace with zero if
true
--  otherwise use corrected data in output.  Hopefully, the synthesis tool can
clean this up

            if(peof = '1')  then
               voutput(18 downto 11) := voutput(18 downto 11);
                    -- if at end of event, leave voutput only but reset vserr;

             elsif(bad_channel_data = '1') then
```

```
              -- bad channel LUT is access using pchip_id and pchannel_id
                   voutput(18 downto 11)   :=  zero8;
                           --  this channel is bad, replace data with zero
                 else
                   voutput(18 downto 11)  := corrected_data;
                            -- not eof and channel good, replace with
                 end if;                                      -- corrected data

              voutput(22 downto 21)   :=  pdata_type;
              voutput(20)             :=  p_new_data_flag;
              voutput(19)             :=  peof;
              voutput(10 downto 7)    :=  pchip_id;
              voutput(6 downto 0)     :=  pchan_id;
              vnew_data_flag   := low;
--
-- save bad channel data for l3
--
              vbc := bad_channel_data;
              next_state := write_fifo;

 --  DO NOT enable l3 raw here
              l3_rawdata_eof <= '0';
              l3_rawdata_enable <= '0';

             -- write_fifo State --
           WHEN write_fifo =>
             vstate := write_fifo_state;   --- this is for debugging
              vnew_data_flag := low;
                      -- reset new data flag after first write.
              if(output_fifo_full = '1') then
                 next_state := write_fifo;
               else
                 next_state := read_fifo;
             -- fifo is free to write into. Next state should be calling state
               if(peof = '1') then     --  if end of event, reset vserr
                  vserr := low;     --
                end if;
             end if;

 --  bring l3_rawdata low in here

             l3_rawdata_eof <= '0';
             l3_rawdata_enable <= '0';

             --read_fifo state
          when read_fifo =>
               vstate := read_fifo_state;
               if(input_fifo_empty = '1') then
                  next_state := read_fifo;  --  fifo is empty.  need to wait
               else
                  next_state := return_state;
               end if;

 --  bring l3_rawdata low in here
               l3_rawdata_eof <= '0';
               l3_rawdata_enable <= '0';



             -- OTHERS --   -- should never come here
           WHEN OTHERS =>
                  next_state := sinit;
                  return_state := sinit;
```

```
                  l3_rawdata_eof <= '0';
                  l3_rawdata_enable <= '0';

          END CASE;

--  if monitor_clear, need to reset monitor counters)
         if(monitor_clear = '1') then
           for i in 8 downto 0 loop
            nchip(i) <= zero_monitor_word;
           end loop;
           nsmt_mismatch_count <= zero24;
           nsmt_zero_error <= zero24;
          end if;

--    set the next state of signal to their respective variable signals
        nstate <= vstate;
        n_new_data_flag <= vnew_data_flag;
        nrs <= return_state;
        ns <= next_state;
        noutput <= voutput;
        neof <= veof;
        nchip_id <= vchip_id;
        nchan_id <= vchan_id;
        nseq_id <= vseq_id;
        nhdi_id <= vhdi_id;
        nbc <= vbc;
        nserr <= vserr;

 end process;
-- The following two process block work concurrently with the above process
block.  One
-- block reads raw data from the input FIFO process block and the other write
corrected data
-- to the write FIFO process block

--  Read fifo block
--  This process block reads raw data from the input fifo

 process(ps,input_fifo_empty)
              --  This process is used to read the input fifo
  variable vread: std_logic;

  begin
    vread := '0';
    if(input_fifo_empty = '1') then
        --  Buffer is empty, therefore no read signal
      vread := '0';
    else
--Write and read fifo process
      case ps is
        when read_fifo =>  -- For any read_fifo state set read flag high
          vread := '1';
        when others =>
          vread := '0';
      end case;
   end if;
   oread <= vread;         -- output read signal is set to read variable
 end process;

  process(ps)        --  This process is used to write to the output fifo
  variable vwrite: std_logic;
  begin
      vwrite := '0';
```

```
    if(output_fifo_full = '1') then      --  If buffer is full, we must wait
      vwrite := '0';
    else
      case ps is         --when we have a write enable signal, we will proceed
        when write_fifo=> -- to write to the ouput fifo the 23-bit word
          vwrite := '1';
        when others =>
          vwrite := '0';
      end case;
    end if;
    owrite <= vwrite;
  end process;
-------------------------------------------------------------------------------
--
---   Use this process block to determine the type of data
---   Input parameters must come from another block
-------------------------------------------------------------------------------
--
process (data_range_in,pchip_id)
--  use variables to break up data_range data
 variable axial_low,axial_high,stereo_low,stereo_high,ninety_low,ninety_high:
unsigned(3 downto 0);
 begin

   axial_low := data_range_in(3 downto 0);
   axial_high := data_range_in(7 downto 4);
   stereo_low := data_range_in(11 downto 8);
   stereo_high := data_range_in(15 downto 12);
   ninety_low := data_range_in(19 downto 16);
   ninety_high := data_range_in(23 downto 20);

   ndata_type <= undefined_type;

   if(pchip_id >= axial_low and pchip_id <= axial_high) then
       ndata_type <= axial_type;
   end if;

   if(pchip_id >= stereo_low and pchip_id <= stereo_high) then
       ndata_type <= stereo_type;
   end if;

   if(pchip_id >= ninety_low and pchip_id <= ninety_high) then
       ndata_type <= ninety_degrees_type;
   end if;
end process;
-------------------------------------------------------------------------------
--
--- This block monitors the number of times ERROR is high in the smt data
-------------------------------------------------------------------------------
--
process(stream_in,monitor_clear)
  begin
    nsmt_error_count <= psmt_error_count;
    if(monitor_clear = '1') then
     nsmt_error_count <= zero24;
    elsif((stream_in(16) = high) or (stream_in(17) = high)) then
     nsmt_error_count <= psmt_error_count + 1;
    end if;
  end process;
---
--- This process block is used to start a new event
---
process(pevent_state,peof,event_start)
```

```
  begin
       nnew_event <= pnew_event;

     case pevent_state is

         when sinit =>
            nnew_event <= '0';
            if(event_start = '1') then
              nevent_state <= snew_event;
            else
              nevent_state <= sinit;
            end if;

         when snew_event =>   -- this is a new event
              nnew_event <= '0';
                     -- enter three wait states before reading first word
              nevent_state <= swait1;  -- this gives L3 time to write headers

         when swait1 =>
              nnew_event <= '0';
              nevent_state <= swait2;

         when swait2 =>
              nnew_event <= '0';
              nevent_state <= sevent;

         when sevent =>
              nnew_event <= '1';
              if(peof = '1') then
               nevent_state <= sinit;
              else
               nevent_state <= sevent;
              end if;
          when others =>    ---  never should come here
              null;
        end case;

end process;



---   Use this process block to clear monitor signals
--------------------------------------------------------------------------------
------------          REGISTER BLOCK      --------------------
--------------------------------------------------------------------------------
--
 reg:process(clock,reset, ns, nrs, noutput,nchip_id,nchan_id,n_new_data_flag,
            ndata_type,neof,nbad_channel_address,nstate,nchip_data,nmismatchbit,
            nsmt_error_count,nsmt_mismatch_count,nsmt_zero_error,nbc,nnew_event)

      begin

       if (reset = '0') then  --Sets all outputs and counters to zero
            pstate         <= initial_state;
            ps             <= sinit;
            prs            <= sinit;
            poutput        <= zero23;
            pchip_id       <= zero4;
            pchan_id       <= zero7;
            p_new_data_flag<= '0';
            pseq_id        <= zero8;
            phdi_id        <= zero3;
```

```
            pdata_type     <= zero2;
            peof           <= '0';
            pbc            <= '0';
            pserr          <= '0';
            pbad_channel_address   <= zero4&zero7;
                    pchip_data      <="000000000000";
            psmt_error_count <= zero24;
            psmt_mismatch_count <= zero24;
            psmt_zero_error <= zero24;
            pl3_trailer <= zero24;
            pmismatchbit <= '0';
            pnew_event <= '0';
            pevent_state <= sinit;

        elsif (clock'event and clock = '1') then
            pstate        <= nstate;
            ps             <= ns;
                      --at every clock edge there will be an output
            prs            <= nrs;
            poutput        <= noutput;
            pchip_id       <= nchip_id;
            pchan_id       <= nchan_id;
            p_new_data_flag<= n_new_data_flag;
            pseq_id        <= nseq_id;
            phdi_id        <= nhdi_id;
            pdata_type     <= ndata_type;
            peof           <= neof;
            pbc            <= nbc;
            pserr          <= nserr;
            pbad_channel_address   <= nbad_channel_address;
            pchip_data     <= nchip_data;
            psmt_error_count <= nsmt_error_count;
            psmt_mismatch_count <= nsmt_mismatch_count;
            psmt_zero_error <= nsmt_zero_error;
            pmismatchbit <= nmismatchbit;
            pl3_trailer <= nl3_trailer;
            pnew_event <= nnew_event;
            pevent_state <= nevent_state;

      end if;

end process reg;               --End of register block

--
-- register block for chip counters
--
process(clock,reset,nchip)
 begin
  if(reset = '0') then

     for i in 8 downto 0 loop
      pchip(i) <= zero_monitor_word;
     end loop;

  elsif(clock'event and clock='1') then
     pchip <= nchip;

  end if;
end process;
--
-- This process block is used to form l3 word
--
```

```
process(peof,pmismatchbit,pdata_type,poutput,pseq_id,phdi_id,pbc,smt_in,pserr,p
l3_trailer)

    variable vnew_data_flag,vend_of_event: std_logic;
    variable vchip_id : unsigned(3 downto 0);
    variable vchan_id: unsigned(6 downto 0);
    variable vdata_event, vseq_id: unsigned(7 downto 0);
    variable vhdi_id: unsigned(2 downto 0);
    variable vdata_type:unsigned(1 downto 0);
    variable veof,vbc,vmismatch,vserr: std_logic;
    variable vsmt_in: unsigned(2 downto 0);
    variable vl3_out: unsigned(24 downto 0);

 begin
--
--  Use variables to help form data word
--

      vdata_type := poutput(22 downto 21);
      vnew_data_flag :=  poutput(20);
      veof := poutput(19);
      vdata_event := poutput(18 downto 11);
      vchip_id := poutput(10 downto 7);
      vchan_id := poutput(6 downto 0);
      veof := peof;
      vmismatch := pmismatchbit;
      vseq_id := pseq_id;
      vhdi_id := phdi_id;
      vsmt_in := smt_in;
      vserr := pserr;
      vbc := pbc;

     nl3_trailer <= vserr&vmismatch&vsmt_in&vseq_id&vhdi_id&vdata_event;

--  if end of event we must output trailer l3 word  --- event number should be
in trailer
      if(veof = '1') then
--                  1  +  1  + 1       + 3  +  8 + 3  + 8  = 25
        vl3_out := veof&vserr&vmismatch&vsmt_in&vseq_id&vhdi_id&vdata_event;
--                       1  +    1    +  3
       else
--                   1+ 1 + 2         + 8   + 4 +      7      + 2 = 25
        vl3_out := veof&vbc&vdata_type&vdata_event&vchip_id&vchan_id&"00";
--        nl3_trailer <= pl3_trailer;
       end if;

       l3_correcteddata <= vl3_out;
  end process;

  ostate <= pstate;           -- Ensures that we keep the outputs constant at
every cycle
  output <= poutput;               -- until a condition is met and we update
the values.
  bad_channel_address <= pbad_channel_address;
  chip_data <= pchip_data;

  chip_monitor <= pchip;
  smt_monitor(0) <= to_monitorword(pl3_trailer);
  smt_monitor(1) <= to_monitorword(psmt_error_count);
  smt_monitor(2) <= to_monitorword(psmt_mismatch_count);
  smt_monitor(3) <= to_monitorword(psmt_zero_error);

end architecture logic_design;   --End of architecture block
```

C.14: MONITOR ADDRESS SPACE

```
--
--  This is the monitor address space
--  It is 32x24
--

library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library cluster_package;
use cluster_package.cluster_package.all;
--  This package contains the following definitions
--  smt_array is 4 x 24
--  chip_array is  9 x 24
--  centroid_array is 3 x 24

entity monitor_spacea is
 port(address: in unsigned(9 downto 0);
      monitor_latch: in std_logic;
      smt_counter: in smt_array;
      chipcounter: chip_array;
      axial:in centroid_array;
      clk,reset: in std_logic;
      l3_trailer: out fifoword;
      word_out: out fifoword;
      monitor_clear: out std_logic
      );
end entity monitor_spacea;

architecture behavior of monitor_spacea is

--
--  Define the following general memory map for the 1k monitor space
--
--  smt counters                    000-00F
--  centroid counters               010-01F
--  chip counters                   020-02F
--  empty space                     030-3FF

constant smt_address_space :unsigned(5 downto 0) := "000000";
constant centroid_address_space :unsigned(5 downto 0) := "000001";
constant chip_address_space :unsigned(5 downto 0) := "000010";


signal nsmt_registers,psmt_registers: smt_array;
signal nchip_registers,pchip_registers: chip_array;
signal ncentroid_registers,pcentroid_registers: centroid_array;

constant true: std_logic := '1';
constant false: std_logic := '0';

signal nmonitor_clear,pmonitor_clear: std_logic;
type smt_states is (s0,s1,s2);
signal psmt,nsmt: smt_states;
signal nword_out, pword_out: fifoword;
--signal nbyte_out, pbyte_out: fifobyte;

begin
```

```
--
-- This process block attaches the monitor counters to the register array
-- It is enabled on the rising edge of monitor_latch
--
process(psmt,smt_counter,chipcounter,axial,monitor_latch)
  begin
--
--  Default outputs
--
--   smt_registers
--
   for i in 3 downto 0 loop
     nsmt_registers(i) <= psmt_registers(i);
    end loop;
--
--  chip registers
--
   for i in 8 downto 0 loop
     nchip_registers(i) <= pchip_registers(i);
    end loop;

--
-- centroid registers
--
   for i in 2 downto 0 loop
     ncentroid_registers(i) <= pcentroid_registers(i);
    end loop;

--
--  look for the rising edge of monitor_latch
--
       case psmt is
        when s0 =>
          nsmt <= s0;
          nmonitor_clear <= '0';
          if(monitor_latch = '1') then
             nsmt <= s1;
          end if;
        when s1 =>
          nsmt <= s2;
          nmonitor_clear <= '1';

          for i in 3 downto 0 loop
           nsmt_registers(i) <= smt_counter(i);
          end loop;

          for i in 8 downto 0 loop
           nchip_registers(i) <= chipcounter(i);
          end loop;

          for i in 2 downto 0 loop
           ncentroid_registers(i) <= axial(i);
          end loop;


        when s2 =>
          nmonitor_clear <= '1';
          if(monitor_latch = '1') then
            nsmt <= s2;
          else
            nsmt <= s0;
          end if;
```

```
        when others =>
           nsmt <= s0;
           nmonitor_clear <= '0';
        end case;
end process;

--
--    This process block sets the data output to the addressed monitor space
--
  process(address,psmt_registers,pchip_registers,pcentroid_registers)
   variable vaddress_high: unsigned(5 downto 0);
   variable vaddress_low: unsigned(3 downto 0);
   variable vword_out: monitor_word;
  begin

     vaddress_high := address(9 downto 4);
     vaddress_low := address(3 downto 0);
     vword_out := zero_monitor_word;

--
--  use address_high to select memory space
--
--  use address_low to select specific word
--
      case vaddress_high is

        when smt_address_space =>
            case vaddress_low is
              when address0 =>
                vword_out := psmt_registers(0);
              when address1 =>
                vword_out := psmt_registers(1);
              when address2 =>
                vword_out := psmt_registers(2);
              when address3 =>
                vword_out := psmt_registers(3);
              when others =>
                vword_out := zero_monitor_word;
             end case;

        when chip_address_space =>

            case vaddress_low is
              when address0 =>
                vword_out := pchip_registers(0);
              when address1 =>
                vword_out := pchip_registers(1);
              when address2 =>
                vword_out := pchip_registers(2);
              when address3 =>
                vword_out := pchip_registers(3);
              when address4 =>
                vword_out := pchip_registers(4);
              when address5 =>
                vword_out := pchip_registers(5);
              when address6 =>
                vword_out := pchip_registers(6);
              when address7 =>
                vword_out := pchip_registers(7);
              when address8 =>
                vword_out := pchip_registers(8);
              when others =>
                vword_out := zero_monitor_word;
```

```
             end case;

         when centroid_address_space =>

             case vaddress_low is
               when address0 =>
                 vword_out := pcentroid_registers(0);
               when address1 =>
                 vword_out := pcentroid_registers(1);
               when address2 =>
                 vword_out := pcentroid_registers(2);
               when others =>
                 vword_out := zero_monitor_word;
               end case;
         when others =>
                 vword_out := zero_monitor_word;
       end case;

    nword_out  <= to_fifoword(vword_out);

 end process;
--
--  This process block implements the registers for the monitor space
--
  process(nsmt_registers,nchip_registers,ncentroid_registers,clk,reset)
   begin
     if(reset='0') then

       for i in 3 downto 0 loop
         psmt_registers(i) <= zero_monitor_word;
        end loop;

       for i in 8 downto 0 loop
         pchip_registers(i) <= zero_monitor_word;
        end loop;

       for i in 4 downto 0 loop
         pcentroid_registers(i) <= zero_monitor_word;
        end loop;

     elsif(clk='1' and clk'event) then

       for i in 3 downto 0 loop
         psmt_registers(i) <= nsmt_registers(i);
        end loop;

       for i in 8 downto 0 loop
         pchip_registers(i) <= nchip_registers(i);
        end loop;

       for i in 2 downto 0 loop
         pcentroid_registers(i) <= ncentroid_registers(i);
        end loop;

    end if;
   end process;

--
--   This process block implements registers
--

   process(nsmt, nmonitor_clear, clk,reset)
    begin
```

```
      if(reset = '0') then
         psmt <= s0;
         pmonitor_clear <= '0';
         pword_out <= zero_fifoword;
      elsif(clk = '1' and clk'event) then
         psmt <= nsmt;
         pmonitor_clear <= nmonitor_clear;
         pword_out <= nword_out;
      end if;
    end process;

--
--  This process block is used to construct l3 trailer
--
   process(smt_counter(0))
     variable vl3temp: fifoword;
    begin

     vl3temp := to_fifoword(smt_counter(0));
  ---                5   +      2              +  3      +  22 =
     l3_trailer <= "11110"&vl3temp(23 downto 22)&"000"&vl3temp(21 downto 0);

    end process;

    monitor_clear <= pmonitor_clear;
    word_out <= pword_out;

end architecture;
```

C.15: CLUSTER PACKAGE

```
-- This code is for the package- for clustering the data for the counters
-- this is just a package declaration common for all the other files

library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

package cluster_package is

 type  monitor_word is array(23 downto 0) of std_logic;
 type  fifoword is array(31 downto 0) of std_logic;
 type  fifobyte is array(7 downto 0) of std_logic;
 type  smt_array is array(3 downto 0) of monitor_word;
 type  chip_array is array(8 downto 0) of monitor_word;
 type  centroid_array is array(2 downto 0) of monitor_word;

 constant zero_monitor_word: monitor_word := "000000000000000000000000";
 constant zero_fifoword: fifoword := "00000000000000000000000000000000";
 constant zero_fifobyte: fifobyte := "00000000";
 constant address0: unsigned(3 downto 0) := "0000";
 constant address1: unsigned(3 downto 0) := "0001";
 constant address2: unsigned(3 downto 0) := "0010";
 constant address3: unsigned(3 downto 0) := "0011";
 constant address4: unsigned(3 downto 0) := "0100";
 constant address5: unsigned(3 downto 0) := "0101";
```

```
 constant address6: unsigned(3 downto 0) := "0110";
 constant address7: unsigned(3 downto 0) := "0111";
 constant address8: unsigned(3 downto 0) := "1000";
 constant address9: unsigned(3 downto 0) := "1001";
 constant addressa: unsigned(3 downto 0) := "1010";
 constant addressb: unsigned(3 downto 0) := "1011";
 constant addressc: unsigned(3 downto 0) := "1100";
 constant addressd: unsigned(3 downto 0) := "1101";
 constant addresse: unsigned(3 downto 0) := "1110";
 constant addressf: unsigned(3 downto 0) := "1111";

--  define data_types
constant undefined_type: unsigned(1 downto 0) := "00";
constant stereo_type : unsigned(1 downto 0)  := "01";
constant axial_type: unsigned(1 downto 0) := "10";
constant ninety_degrees_type: unsigned(1 downto 0) := "11";


function to_fifoword(a: fifobyte) return fifoword;
function to_fifoword(a: monitor_word) return fifoword;
function to_monitorword(a: unsigned(23 downto 0)) return monitor_word;
function add_one(a: monitor_word) return monitor_word;

end package;

package body cluster_package is
 function to_fifoword(a:fifobyte) return fifoword is
    variable result: fifoword;
   begin
        for i in 31 downto 8 loop
          result(i) := '0';
        end loop;
        for i in 7 downto 0 loop
          result(i) := a(i);
        end loop;
        return result;
   end function; --

 function to_fifoword(a:monitor_word) return fifoword is
    variable result: fifoword;
   begin
        for i in 31 downto 24 loop
          result(i) := '0';
        end loop;
        for i in 23 downto 0 loop
          result(i) := a(i);
        end loop;
        return result;
   end function; --

 function to_monitorword(a:unsigned(23 downto 0)) return monitor_word is
    variable result: monitor_word;
   begin
        for i in 23 downto 0 loop
          result(i) := a(i);
        end loop;
        return result;
   end function; --


function add_one(a:monitor_word) return monitor_word is
    variable result: monitor_word;
    variable temp: unsigned(23 downto 0);
```

```
    variable i: integer;
   begin
        for i in 23 downto 0 loop
          temp(i) := a(i);
        end loop;
          temp := temp + 1;
        for i in 23 downto 0 loop
          result(i) := temp(i);
        end loop;
        return result;
   end function; --
end;
```

C.16: CLUSTER FINDER

```
--------------------------------------------------------------------------------
-- Revision 2 Cluster Finder Algorithm
-- Based on 2-15-2000 Specification Document
--------------------------------------------------------------------------------
-- Initial Design : Reginald Perry (6-12-2000)
--------------------------------------------------------------------------------
--  Add monitor counter  - 7/17/2000 RJP
--  Add end of event bit - 7/17/2000 RJP

library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library cluster_package;
use cluster_package.cluster_package.all;


entity cluster_finder is
    port(data_in: in unsigned(22 downto 0);
              --  input data word from fifo
         data_threshold_1,data_threshold_2: in unsigned(7 downto 0);
              -- read as inputs
         cluster_type: in unsigned(0 downto 0);
              -- use 3 or 5 strips in scalculation
--
--   altera compiler does not like unsigned without vector.
--   Need to have two bits until
--   we can check this out
         input_fifo_empty,output_axialfifo_full,clk, reset : in std_logic;
         input_fifo_read,output_axialfifo_write: out std_logic;
         ad2: out unsigned(10 downto 0); -- address bus
         d1,d2,d3,d4,d5: out unsigned(7 downto 0);
         current_type: out unsigned(1 downto 0);  -- current data type
         end_of_event_out: out std_logic;  -- this is the end of event bit
         calculation_type: out unsigned(0 downto 0);
              -- this is the type of calculation
         l3_enable: out std_logic; -- this is used to enable l3
         centroid_counters: out centroid_array;
         l3out:out unsigned(31 downto 0)
        );

--   inputs for data_threshold
end entity cluster_finder;
```

```
architecture behavior of cluster_finder is

--  define major states
type mystates is (sreset, sread,sread_fifo,sload_fifo,scheck_for_eof,
                  sinit,smain,snext,soutpeak,scalculate,swrite,swrite_fifo,
                  sl3_wait);
    type l3states is (s0,s1,s2,s3,s4,s5,s6);


--  data types are now defined in cluster_package

--    constant axial: unsigned(1 downto 0) := "01";
--    constant stereo: unsigned(1 downto 0) := "10";
--    constant z_axis: unsigned(1 downto 0) := "11";
--    constant undefined: unsigned(1 downto 0) := "00";

    signal ns, ps,nreturn_state,preturn_state: mystates;
    signal nl3s,pl3s: l3states;

--  counter used to keep track of number of strips read
    signal nstrip_count, pstrip_count: unsigned(7 downto 0);

--  signals used for data and address registers
    signal nd1,nd2,nd3,nd4,nd5,ndata_shadow1,
           ndata_shadow2: unsigned(7 downto 0);
    signal pd1,pd2,pd3,pd4,pd5,pdata_shadow1,
           pdata_shadow2: unsigned(7 downto 0);
    signal nad1,nad2,nad3,nad4,nad5,naddress_shadow1,
           naddress_shadow2: unsigned(10 downto 0);
    signal pad1,pad2,pad3,pad4,pad5,paddress_shadow1,
           paddress_shadow2: unsigned(10 downto 0);

--  these signals are used to keep track of the current data type
    signal nnext_address,pnext_address: unsigned(10 downto 0);
    signal ncurrent_type,pcurrent_type: unsigned(1 downto 0);
    signal nfirst_read,pfirst_read,data_compare,valid_cluster_peak: std_logic;
    signal pdata,ndata: unsigned(22 downto 0);

--   these signals are for the end of event bit
    signal neof,peof,nl3_wait,pl3_wait: std_logic;

---  these signals are used for monitoring

    signal naxial_counter,paxial_counter,nstereo_counter,
           pstereo_counter: unsigned(23 downto 0);
    signal nninety_counter, pninety_counter: unsigned(23 downto 0);

--  l3buffer
    signal pl3out,nl3out: unsigned(31 downto 0);

--  these constants are used with the data registers
 constant zero_data_reg: unsigned(7 downto 0) := "00000000";
 constant zero_address_reg: unsigned(10 downto 0) := "00000000000";
 constant zero23:unsigned(22 downto 0) := "00000000000000000000000";
 constant zero24:unsigned(23 downto 0) := "000000000000000000000000";
 constant zero32: unsigned(31 downto 0) := "00000000000000000000000000000000";
 constant true: std_logic := '1';
 constant false: std_logic := '0';

--

--   l3stuff
```

```
 begin

   process(ps,preturn_state,pd1,pd2,pd3,pd4,pd5,pad1,pad2,pad3,pad4,pad5,
           paddress_shadow1,paddress_shadow2,pdata_shadow1,pdata_shadow2,
           pnext_address,pcurrent_type,pdata,preturn_state,pfirst_read,pdata,
           pstrip_count,data_compare,valid_cluster_peak,data_threshold_1,
           data_threshold_2,input_fifo_empty,cluster_type,paxial_counter,
           pstereo_counter,pninety_counter,pl3_wait)

    variable vtype: unsigned(1 downto 0);
    variable veof,vnew_data_flag,end_of_cluster: std_logic;
    variable vdata: unsigned(7 downto 0);
    variable vaddress: unsigned(10 downto 0);
    variable vl3out:unsigned(31 downto 0);

    begin

--   set default outputs

--   state information
     ns <= ps;
     nreturn_state <= preturn_state;   --  hold onto return state

--   save data buffers
     nd1 <= pd1;
     nd2 <= pd2;
     nd3 <= pd3;
     nd4 <= pd4;
     nd5 <= pd5;
     ndata_shadow1 <= pdata_shadow1;
     ndata_shadow2 <= pdata_shadow2;


--  save address buffers
     nad1 <= pad1;
     nad2 <= pad2;
     nad3 <= pad3;
     nad4 <= pad4;
     nad5 <= pad5;
     naddress_shadow1 <= paddress_shadow1;
     naddress_shadow2 <= paddress_shadow2;

--   parse current input word in parts. Use variables for this.

      vtype := pdata(22 downto 21);
      vnew_data_flag := pdata(20); -- variable defined as std_logic
      veof := pdata(19);           -- variable defined as std_logic
      vdata := pdata(18 downto 11);
      vaddress := pdata(10 downto 0);
--  save eof for next block
      neof <= veof;
--
--   set defaults for these signals
--
     ncurrent_type <= pcurrent_type;    --  current data type
     nnext_address <= pnext_address;
     nfirst_read <= pfirst_read;
          --  flag to indicate we are coming out of a reset state
          --  this prevents a centriod calculation before the first read.
     ndata <= pdata;    -- save data

--    This implements the main FSM
```

```
     case ps is

--
--   Reset state
--   This is the first state AFTER the initial reset
--
       when sreset =>
--
--    Need to load data parameters here eventually.
--    For now, just go to the first read
--
         ns <= sread;
         ncurrent_type <= axial_type;
          -- default to axial type to give memory something to access
         nreturn_state <= sinit;
          -- this statement is really not needed since this is the first read
         nfirst_read <= true;     -- but let's help the synthesis tool anyway.

--  Read state
         when sread =>
             if(input_fifo_empty = true) then
                ns <= sread;
                  --  fifo is empty. Need to wait for some data
             else
                ns <= sread_fifo;
                  --  data is ready  Go to next state to read_fifo
             end if;

--   Read fifo state
--   add this "wait" state to allow data to be read from input fifo
          when sread_fifo =>
           ns <= sload_fifo;
--
--    Data should be available at fifo.  Load into an internal register data
--
          when sload_fifo =>
             ndata <= data_in;
             ns <=  scheck_for_eof;  -- let's go check for the eof
--
--     probably we can combine sread_fifo and sload_fifo.  Look at this later
--
--  This state checks for an eof
--

         when scheck_for_eof =>

-- if veof=1 then this is the end of a cluster
-- if veof=0 then end of cluster is initial reset to 0

         end_of_cluster :=veof;
--
-- Check for a change of type
         if(vtype /= pcurrent_type) then
              end_of_cluster := true;
              ncurrent_type <= vtype;   -- save new data type as next
current_type
         end if;

-- if data is less than threshold then end of cluster.  Data_threshold_1 is an
input
         if(vdata < data_threshold_1) then
             end_of_cluster := true;
```

```
        end if;

-- if current_address /= next_address then end of cluster
-- need to set next_address based on this outcome

        if(vaddress /= pnext_address) then
            end_of_cluster := true;
            nnext_address <= vaddress + 1;
        else
            nnext_address <= pnext_address + 1;
        end if;

--
--   if this is the first read then we don't want to store
--   a hit in the output fifo
--
        if(pfirst_read = true) then
            nfirst_read <= false;
            ns<= sinit;
        elsif(end_of_cluster = '1') then
--
--      This is the start of a new cluster.
--      Need to calculate centroid on old cluster.
--      The current data word will be used in the new cluster
--
            ns <= scalculate;
            nreturn_state <= sinit;
      -- after calculate need to initialize current data word
        else
            ns <= preturn_state;
      --  this is not the end of a cluster.  continue cluster finding algorithm
        end if;


--  Initialization State.  We are starting the search for a new cluster

        when sinit =>
-- init data registers
            nd1 <= zero_data_reg;
            nd2 <= zero_data_reg;
            nd3 <= vdata;
            nd4 <= zero_data_reg;
            nd5 <= zero_data_reg;

-- init address registers
            nad1 <= zero_address_reg;
            nad2 <= zero_address_reg;
            nad3 <= vaddress;
            nad4 <= zero_address_reg;
            nad5 <= zero_address_reg;

-- init shadow registers
            ndata_shadow2 <= zero_data_reg;
            ndata_shadow1 <= zero_data_reg;
            naddress_shadow2 <= zero_address_reg;
            naddress_shadow1 <= zero_address_reg;

--  go read the next word and return to smaim
            ns <= sread;
            nreturn_state <= smain;

--   this is the main state.  Compare the next data word to the current cluster
center
```

```
        when smain =>
--   next state will be sread
          ns <= sread;

          if(data_compare = '1') then
--
--  if data  is greater than or equal to current peak then move data as cluster
peak
--  need to shift all other data words to the left by one slot

            nd1 <= pd2;
            nd2 <= pd3;
            nd3 <= vdata;

--  need to also shift addresses

            nad1 <= pad2;
            nad2 <= pad3;
            nad3 <= vaddress;
            nreturn_state <= smain;    --  come back to this state
         else
--   Data is less than current cluster peak.  Store this value in d4.

            nd4 <= vdata;
            nad4 <= vaddress;
            nreturn_state <= snext;  ---  need to go to snext state
        end if;

--
--   this is the next state.  This looks at the D5 register
--
        when snext =>

--  next state will be sread

          ns <= sread;
          if(data_compare = '1') then
--  if data is greater than or equal to current cluster peak
--  shift d3 to d1 and d4 to d2 and new peak (nd3) gets vdata
--  must also remember to zero out nd4 and nd5
            nd1 <= pd3;
            nd2 <= pd4;
            nd3 <= vdata;
            nd4 <= zero_data_reg;
            nd5 <= zero_data_reg;

--   also need to change the addresses

            nad1 <= pad3;
            nad2 <= pad4;
            nad3 <= vaddress;
            nad4 <= zero_address_reg;
            nad5 <= zero_address_reg;

            nreturn_state <= smain;
         else

--   vdata  is less than current cluster peak
--   save vdata in nd5 also need to set up "shawdow" registers
            nd5 <= vdata;
            nad5 <= vaddress;
```

```
--   need to setup shadow registers for "out-peak" case
             ndata_shadow1 <= pd4;
             ndata_shadow2 <= vdata;
             naddress_shadow1 <= pad4;
             naddress_shadow2 <= vaddress;
--   return state should be out peak
             nreturn_state <= soutpeak;

         end if;

--  This is the outpeak case.
--  That is, the current cluster address is out of the range of the
--  cluster peak.  However, we need to save the last two values in case
--  we come across a case where
--  vdata > pd3.

         when soutpeak =>

--  next state will be sread

            ns <= sread;
--
-- This is where we check the data against the current cluster peak
--
            if(data_compare = '1') then

-- data is greater than current cluster peak.
-- Need to move shadow registers into real registers
-- don't forget to zero d4 and d5

             nd1 <= pdata_shadow1;
             nd2 <= pdata_shadow2;
             nd3 <= vdata;
             nd4 <= zero_data_reg;
             nd5 <= zero_data_reg;

--   same goes for the addresses

             nad1 <= paddress_shadow1;
             nad2 <= paddress_shadow2;
             nad3 <= vaddress;
             nad4 <= zero_address_reg;
             nad5 <= zero_address_reg;

             nreturn_state <= smain;   --- return now goes to smaim
           else
--
--  din is not greater than current peak.
--  shift shadow registers and return to this state
--
            ndata_shadow1 <= pdata_shadow2;
            ndata_shadow2 <= vdata;
            naddress_shadow1 <= paddress_shadow2;
            naddress_shadow2 <= vaddress;
            nreturn_state <= soutpeak;
           end if;

--
--  This is the centriod calculate state
--  Actually centroid is always being calculated
--  we just need to determine if we send it on to the
--  hit filter.
--
```

```
           when scalculate =>
--
--
--
           if(valid_cluster_peak = true) then   --- this signal is generated by
another process block
--
--          need to and increment counters
            case vtype is
             when axial_type =>
                naxial_counter <= paxial_counter + 1;
             when stereo_type =>
                nstereo_counter <= pstereo_counter + 1;
             when ninety_degrees_type =>
                nninety_counter <= pninety_counter + 1;
             when others =>
                 null;
            end case;


            ns <= swrite;
           else
--
--   this is not a valid peak.  Throw away this cluster and start over
--
            ns <= sinit;
           end if;

           when swrite =>
--   dummy state to see if output fifo is not empty
           if(output_axialfifo_full = true) then
              ns <= swrite;
           else
              ns <= swrite_fifo;
           end if;

           when swrite_fifo =>
-- output fifo is written during this cycle
            ns <= sl3_wait;   --  let's wait for l3 to finish

           when sl3_wait =>
             if(pl3_wait = '1') then
               ns <= sl3_wait;
             else
               ns <= sinit;
             end if;

           when others =>
            -- we should never come here.  Therefore go to sreset
             ns <= sreset;

        end case;
   end process;

--
--  this state machine is used to read a word from the input fifo
--

process(ps)
  variable vread: std_logic;
  begin
      vread := false;
```

```
      case ps is
       when sread_fifo =>
         vread := true;
       when others =>
         vread := false;
      end case;
       input_fifo_read <= vread;
  end process;

--
--  this state machine is used to write a word to the output fifo
--

process(ps)
  variable vwrite: std_logic;
  begin
      vwrite := false;
      case ps is
       when swrite_fifo =>
         vwrite := true;
       when others =>
         vwrite := false;
      end case;
       output_axialfifo_write<= vwrite;
              -- do this as a mealy machine to save a cycle
  end process;

--
--   This process block counts the number of strips in the current cluster (-1)
--
process(ps,pstrip_count)
 variable vcnt : unsigned(7 downto 0);
begin
    vcnt := pstrip_count;
    case ps is
      when sinit =>
        vcnt := zero_data_reg;
      when smain|snext|soutpeak =>
            -- only increment this counter when we have new data
        vcnt := vcnt + 1;
      when others =>
        null;
     end case;
    nstrip_count <= vcnt;
 end process;



--
--   This process block compares the data against the current peak
--

   process(pdata,pd3)
    variable vdata:unsigned(7 downto 0);
   begin
    vdata := pdata(18 downto 11);
--
--  compare data word to current cluster center
--
    if(vdata < pd3) then
      data_compare <= '0';
    else
      data_compare <= '1';
```

```
    end if;
   end process;

--
--   This process block compares pd3 against data_threshold_2
--

   process(pd3,data_threshold_2)
    variable vdata:unsigned(7 downto 0);
   begin
    vdata := pd3;
--
--  compare data word to current cluster center
--
    if(vdata < data_threshold_2) then
      valid_cluster_peak <= false;
              -- current peak is less than data_threshold_2 -- invalid peak.
    else
      valid_cluster_peak <= true;   -- we have a valid peak here.
    end if;
   end process;

--
--  Register process block for state information
--
 process (ns,nreturn_state,nl3s,clk, reset)
   begin
    if( reset = '0') then
      ps <= sreset;
      preturn_state <= sreset;
      pl3s <= s0;
    elsif(clk = '1' and clk'event) then
      ps <= ns;
      preturn_state <= nreturn_state;
      pl3s <= nl3s;
   end if;
  end process;
--
--  Register process block for misc signals
--
 process (ncurrent_type,nnext_address, pfirst_read, clk, reset)
   begin
    if( reset = '0') then
      pcurrent_type <= undefined_type;
      pnext_address <= zero_address_reg;
      pfirst_read <= '0';
    elsif(clk = '1' and clk'event) then
      pcurrent_type <= ncurrent_type;
      pnext_address <= nnext_address;
      pfirst_read <= nfirst_read;
   end if;
  end process;
--
--  Register process block to shadow address and data registers also the data
is saved here
--
  process(nd1,nd2,nd3,nd4,nd5,nad1,nad2,nad3,nad4,nad5,nstrip_count,
          nl3out,nl3_wait,naddress_shadow1,naddress_shadow2,
          ndata_shadow1,ndata_shadow2,ndata, clk, reset)
   begin
    if( reset = '0') then
       pd1 <= zero_data_reg;
       pd2 <= zero_data_reg;
```

```
        pd3 <= zero_data_reg;
        pd4 <= zero_data_reg;
        pd5 <= zero_data_reg;
        pad1 <= zero_address_reg;
        pad2 <= zero_address_reg;
        pad3 <= zero_address_reg;
        pad4 <= zero_address_reg;
        pad5 <= zero_address_reg;
        pdata_shadow1 <= zero_data_reg;
        pdata_shadow2 <= zero_data_reg;
        paddress_shadow1 <= zero_address_reg;
        paddress_shadow2 <= zero_address_reg;
        pdata <= zero23;
        pstrip_count <= zero_data_reg;
        pl3out <= zero32;
        peof <= '0';
        pl3_wait <= '0';

      elsif( clk'event and clk = '1') then

        pd1 <= nd1;
        pd2 <= nd2;
        pd3 <= nd3;
        pd4 <= nd4;
        pd5 <= nd5;
        pad1 <= nad1;
        pad2 <= nad2;
        pad3 <= nad3;
        pad4 <= nad4;
        pad5 <= nad5;
        pdata_shadow1 <= ndata_shadow1;
        pdata_shadow2 <= ndata_shadow1;
        paddress_shadow1 <= naddress_shadow1;
        paddress_shadow2 <= naddress_shadow1;
        pdata <= ndata;
        pstrip_count <= nstrip_count;
        pl3out <= nl3out;
        peof <= neof;
        pl3_wait <= nl3_wait;

      end if;
    end process;

--  Registers for counters
     process(clk,reset,naxial_counter,nstereo_counter,nninety_counter)
      begin
        if(reset = '0') then
          paxial_counter <= zero24;
          pstereo_counter <= zero24;
          pninety_counter <= zero24;
        elsif(clk'event and clk='1') then
          paxial_counter <= naxial_counter;
          pstereo_counter <= nstereo_counter;
          pninety_counter <= nninety_counter;
        end if;
     end process;

--  this process block takes care of l3, we will need multiple cycles to write
all the data

process(ps,
pl3s,pd1,pd2,pd3,pd4,pd5,pad1,pad2,pad3,pad4,pad5,data_threshold_1,data_thresho
ld_2,pstrip_count,
```

```
        valid_cluster_peak, pcurrent_type, cluster_type)

       variable vl3out: unsigned(31 downto 0);
 begin
-- default outputs

            nl3_wait <= '0';
            vl3out := zero32;
            nl3s <= pl3s;
            case pl3s is

              when s0 =>
                  l3_enable <= '0';
                if(ps = scalculate) then
                  nl3_wait <= '1';  --- tell main fsm to hold
                  nl3s <= s1;
--  need to set up first l3 output word
                  vl3out(31 downto 28) := "0000";
                  vl3out(27 downto 20) := data_threshold_2;
                  vl3out(19 downto 18) := pcurrent_type;
                  vl3out(17) := cluster_type(0);     --- only a single bit
                  vl3out(16 downto 9) := data_threshold_1;
                  vl3out(8 downto 1) := pstrip_count;
                  vl3out(0) := '1';
                else
                  nl3_wait <= '0';
                  nl3s <= s0;
                end if;

               when s1  =>
                  nl3_wait <= '1';
                  l3_enable <= '1';  -- write the previous word into l3 using
l3_enable --do not register
                  vl3out(31 downto 23) := "000000000";
                  vl3out(22 downto 21) := pcurrent_type;
                  vl3out(20 downto 13) := pd2;
                  vl3out(12 downto 2) := pad2;
                  vl3out(1 downto 0) := "00";
                  nl3s <= s2;

                when s2 =>
                  nl3_wait <='1';
                  l3_enable <= '1';
---  write the remaining data out
                  vl3out(31 downto 23) := "000000000";
                  vl3out(22 downto 21) := pcurrent_type;
                  vl3out(20 downto 13) := pd3;
                  vl3out(12 downto 2) := pad3;
                  vl3out(1 downto 0) := "00";
                  nl3s <= s3;

                when s3 =>
                  nl3_wait <='1';
                  l3_enable <= '1';
---  write the remaining data out
                  vl3out(31 downto 23) := "000000000";
                  vl3out(22 downto 21) := pcurrent_type;
                  vl3out(20 downto 13) := pd4;
                  vl3out(12 downto 2) := pad4;
                  vl3out(1 downto 0) := "00";
                  nl3s <= s4;

                when s4 =>
```

```
                  nl3_wait <='1';
                  l3_enable <= '1';
---  write the remaining data out
                  vl3out(31 downto 23) := "000000000";
                  vl3out(22 downto 21) := pcurrent_type;
                  vl3out(20 downto 13) := pd5;
                  vl3out(12 downto 2) := pad5;
                  vl3out(1 downto 0) := "00";
                  nl3s <= s5;

                when s5 =>
                  nl3_wait <='0';      --  we can let main FSM restart now
                  l3_enable <= '1';
---  write the remaining data out
                  vl3out(31 downto 23) := "000000000";
                  vl3out(22 downto 21) := pcurrent_type;
                  vl3out(20 downto 13) := pd1;
                  vl3out(12 downto 2) := pad1;
                  vl3out(1 downto 0) := "00";
                  nl3s <= s6;    --  need one more state to write out last word

                when s6 =>
                  nl3_wait <='0';
                  l3_enable <= '1';       --- write out last word
                  nl3s <= s0;
                when others =>
                  null;
              end case;
            nl3out <= vl3out;  -- this will save l3out for writing into l3

 end process;

--   attach address and data registers to output buffers
     d1 <= pd1;
     d2 <= pd2;
     d3 <= pd3;
     d4 <= pd4;
     d5 <= pd5;
     ad2 <= pad2;
     l3out <= pl3out;
     current_type <= pcurrent_type;
     end_of_event_out <= peof;
     calculation_type <= cluster_type;

--  attach counters
     centroid_counters(0) <= to_monitorword(paxial_counter);
     centroid_counters(1) <= to_monitorword(pstereo_counter);
     centroid_counters(2) <= to_monitorword(pninety_counter);



 end behavior;
```

C.17: CENTROID CALCULATOR

```
----------------------------------------------------------------
--  File: centroid_calculator.vhd
--  Description:  VHDL code to calculate the three or five strip centroid.
--  For the five strip case we are given five (5) eight bit values -
-- d1,d2,d3,d4,d5
```

```
--  The centroid is usually given by  (d1+2d2+3d3+4d4+5d5)/(d1+d2+d3+d4+d5)
--  in this case we are shifting the result so it is centered around d2
--  so the numerator becomes   -d1+0d2+d3+2d4+3d5
--  For the three strip case: the equation is -d1+0d2+d3/d1+d2+d3
--
--  Originally Created:   6/4/00
-- (Reginald J. Perry - FAMU-FSU College of Engineering)
-- Add l3 logic:  7/17/2000   RJP
-- Modified : by Shweta Lolage
-- Changes in the 3 cluster centroid calculations
----------------------------------------------------------------
--- normal initialization stuff

library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

--
--  define entity
--
entity centroid_calculator is
--
--  The maximum value is given by 5*255 which will need 8+3+1=12 bits to save -
-- (need  one bit for the sign).  We'll assume the d's are unsigned values.
--  Therefore, we will need N+4 bits for the num and den results.

  port(d1,d2,d3,d4,d5: in signed(7 downto 0);
       threshold_range_in:in unsigned(35 downto 0); -- this is threshold data
       address_d2: in signed(10 downto 0);
              -- this is the original address of d2
       calculation_type: in std_logic;
       centroid_type: in signed(1 downto 0);
       end_of_event_bit: in signed(0 downto 0);
       l3_trailer: in signed(31 downto 0);
       l3_out: out signed(31 downto 0);
       centroid_out: out signed(17 downto 0)
      );
end entity centroid_calculator;

architecture behavior of centroid_calculator is

signal  num,den: signed(11 downto 0);
constant three_strip_centroid: std_logic := '0';
constant five_strip_centroid: std_logic := '1';
constant high: std_logic := '1';
constant low: std_logic := '0';
constant zero9 : signed(8 downto 0) := "000000000";

-- these signals are needed by the divider circuit
--
signal q,qint: signed(3 downto 0);
signal y3,y3m,y2,y2m,y1,y1m: signed(11 downto 0);
signal centroid,sum3,sum2,sum1,sum0: signed(12 downto 0);

--  These signals and constants are used to calculate the quantized pulse areas

signal centroid_pulse_area: signed(1 downto 0);
signal level1_check,level2_check,level3_check: std_logic;
signal level_check: std_logic_vector(2 downto 0);
constant level0 : signed(1 downto 0) := "00";
constant level1 : signed(1 downto 0) := "01";
```

```
constant level2 : signed(1 downto 0) := "10";
constant level3 : signed(1 downto 0) := "11";

begin

--   This process block provides the sums for the numerator and denominator

summer: process(d1,d2,d3,d4,d5,calculation_type)
--  Create variables to temporarily save results

  variable s1,s2,s3,s4,s5           : signed(8 downto 0);
  variable temp1,temp1a,temp2,temp3 : signed(9 downto 0);
  variable temp32,temp5             : signed(10 downto 0);
  variable parea,temp4,temp6,temp7  : signed(11 downto 0);
  variable temp1b,temp2b            : signed (9 downto 0);
  variable temp3b                   : signed (10 downto 0);
  begin
--
--  s's are used to add a sign bit to the d's
--
    s1 := "0"&d1;
    s2 := "0"&d2;
    s3 := "0"&d3;
    s4 := "0"&d4;
    s5 := "0"&d5;

--  define temporary variables needed by both the three and five strip
calculator

     temp1 := s1+s2;          -- this sum will be 10 bits (d1+d2)
     temp2 := s3-s1;          -- this sum will be 10 bits (d3-d1)

--  Use two separate adders for three and five strip centroid calculation
--  this may save some time in the calculation

    if(calculation_type = five_strip_centroid) then

     temp3 := s4+s5;          -- this sum will be 10 bits (d4+d5)
     temp32 := temp3&"0";     -- this is 2xtemp3 = 2(d4+d5); it will be 11 bits

     temp4 := ("00"&s5)+temp32;    -- this is 2d4 + 3d5: sum will be 12 bits
     temp5 := temp1 + temp3;       -- this is d1+d2+d4+d5: sum will be 11 bits

     den <= ("000"&s3)+temp5;     -- this is d1+d2+d3+d4+d5: need 12 bits
     temp7 := "00"&temp2;         -- bit extend temp2 from 10 to 12 bits
     num <=  temp7+temp4;         --  this is -d1+d3+2d4+3d5:  need 12 bits
    else
-- this is the three strip case. num is -d2+d4 and den = d2+d3+d4
        temp1b := s4 - s2;  -- this will be num = -d2+d4
      num <= "000"&temp1b;    -- sign extend to 12 bits
        temp2b := s2 + s4;  -- the sum of d2 + d4 , for the den
      temp1a := "00"&d3;   --  need to sign extend d3 to add it to temp1
      den <= "0"&(temp2b+temp1a);
    end if;

   end process summer;

--   These process blocks implements the divider circuit.

--   The design consists of four identical process blocks with a 2x1 mux
--   Each process calculates   yx - b  where yx the result from the previous
--   process block.  For the first process block, yx=a.  If the result of yx-a
-- is greater than 0 (i.e. positive)
```

```
-- then this result is given to the next process
--   block otherwise  yx is propagated.
-- A mux is used to select between the two possiblities.
--
--

p3:process(num,den)

  begin
    sum3 <= num - den;

--
--  the "sign bit" is stored in sum(12) and saved as qint(3)
--
    qint(3) <= sum3(12);
  end process;

--
--  if qint(3) is 1, then the result is negative and we DONT want to keep the
subtraction result
--  in this case we'll send yx (num in this case) to the next process block.
If qint = 0 , then
--   yx >= b and we want to keep the subtraction.
--
with qint(3) select
   y3m   <=  num                 when '1',
             sum3(11 downto 0)   when '0',
             num                   when others;

--
--  We need to shift the result from above to the left by one bit so we can
calculate the next
--  bit of the division.  We could also accomplish this by shifting den one bit
to the right.
--
   y3 <= y3m(10 downto 0) & '0';

--
--  This process block is identical to the above except num has been
--  replaced with y3 and
--  the appropriate substitutions have been made for sum, y, and qint.
--
p2:process(y3,den)
  begin
    sum2 <= y3 - den;
    qint(2) <= sum2(12);
  end process;

with qint(2) select
   y2m  <=  y3               when '1',
            sum2(11 downto 0) when '0',
            y3                 when others;

   y2 <= y2m(10 downto 0) & '0';

--
--  This process block is identical to the above except y3
--  has been replaced with y2 and
--  the appropriate substitutions have been made for sum, y, and qint.
--

p1:process(y2,den)
  begin
```

```
    sum1 <= y2 - den;
    qint(1) <= sum1(12);
  end process;

with qint(1) select
   y1m  <=  y2               when '1',
            sum1(11 downto 0) when '0',
            y2                when others;

   y1 <= y1m(10 downto 0) & '0';
--
--  This process block is identical to the above except
--  y2 has been replaced with y1 and
--  the appropriate substitutions have been made for sum, y, and qint.
--

p0:process(y1,den)
  begin

    sum0 <= y1 - den;
    qint(0) <= sum0(12);
  end process;

inv:for I in 3 downto 0 generate
   q(i) <= not qint(i);
    end generate inv;

--  q's now contain the result of the division.
--  Need to add this result to the original address of d2

   process(address_d2,q)
     variable tempa, tempb: signed(12 downto 0);
     constant zero10: signed(9 downto 0) := "0000000000";
    begin
      -- the result should be 13 bits
      -- original 11 bits plus two extra bits
      -- let's use a couple of variables so we don't get lost
      tempa := address_d2&"00";
      tempb := zero10&q(3 downto 1);
      centroid <= tempa + tempb;

    end process;

--  This process block will calculate the two bit pulse area threshold values
--------------------------------------------------------------------------------
--
--------------------------------------------------------------------------------
--
process (threshold_range_in,den)
--  use variables to break up data_range data
 variable level1_low,level2_low,level3_low:unsigned(11 downto 0);
 variable parea:signed(1 downto  0);
 constant code_level0:std_logic_vector(2 downto 0)  := "000";
 constant code_level1: std_logic_vector(2 downto 0) := "001";
 constant code_level2: std_logic_vector(2 downto 0) := "011";
 constant code_level3 : std_logic_vector(2 downto 0) := "111";

 begin

--  implement this like a coarse approximation ADC
--  Check input data against threshold levels and then use an encoder to
--  convert thermometer code to binary code
```

```
   level1_low := threshold_range_in(11 downto 0);
   level2_low := threshold_range_in(23 downto 12);
   level3_low := threshold_range_in(35 downto 24);

-- make the default level 3

   level1_check <= high;
   level2_check <= high;
   level3_check <= high;

   if(den < level1_low) then
     level1_check <= low;
   end if;

   if(den < level2_low) then
     level2_check <= low;
   end if;

   if(den < level3_low) then
     level3_check <= low;
   end if;

    level_check <= level3_check&level2_check&level1_check;

   case level_check is
      when code_level0 =>
         parea := level0;
      when code_level1 =>
         parea := level1;
      when code_level2 =>
         parea := level2;
      when code_level3 =>     -- this is the default condition
        parea := level3;
      when others =>
        parea := level3;
     end case;

 centroid_pulse_area <= parea;
end process;

-- this process block is for l3 for the centroids
process(l3_trailer,address_d2,centroid_pulse_area,centroid_type,end_of_event_bi
t)
   variable vsmt,vhdi_id: signed(2 downto 0);
   variable vseq_id: signed(7 downto 0);
 begin

   vsmt := l3_trailer(21 downto 19);
   vhdi_id := l3_trailer(10 downto 8);
   vseq_id := l3_trailer(18 downto 11);

   if(end_of_event_bit(0) = '1') then
     l3_out <= l3_trailer;
   else
--              1+  3  +  2          +     2             +  8    + 3     +  13
=
     l3_out <=
"0"&vsmt&centroid_type&centroid_pulse_area&vseq_id&vhdi_id&centroid;
   end if;
 end process;
```

```
--  centroid out is the end_of_event, datatype, centroid_pulse_area and
centroid
--                          1         +     2       +     2             +  13
 centroid_out <=
end_of_event_bit(0)&centroid_type&centroid_pulse_area&centroid;


end architecture behavior;
```

C.18: HIT CONTROL BLOCK

```
-------------------------------------------------------------------
--File:hit_control_block.vhd
-- To implement the control logic for the hit filter.
-- Originally created by : Shweta Lolage date - 05/15/2000
--Modifications by Shweta : - 06/02/2000
-- modified by Shweta lolage on 06/15/2000 to remove all unnecessary registered
signals
-- modified to include one state to accommodate a control signal to read the
masked register
-- by Shweta Lolage - 09/04/2000
-------------------------------------------------------------------

library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;


--  Entity Declaration
entity hit_control_block is
       port (road_write,done,event_start : in std_logic;
              -- road_write signal from the road data handler
             -- DONE signal is from the pointer block indicating
             --that it is through with the hits for that particular centroid.
             -- event _start is a global signal marking the start of an event
            clock,reset : in std_logic;
            last_road : in std_logic; --signal indicating end of road data
             fifoempty : in std_logic;   -- active low signal
             datatype  : in std_logic_vector (2 downto 0);
       -- the end of event tagged with the datatype of the centroid
       -- coming from the centroid finder
            centroid  : in unsigned (12 downto 0);
                 -- centroids from the centroid finder
           road,mask : out unsigned (5 downto 0);
-- the signals to the ROMs to get the corrected load road signal
-- and final mask register
          read_comp,read_hitreg,input_fifo_enable,hits_busy : out std_logic;
             --the read_comp signal is to read the output from the comparators
             -- this read_hitreg signal is to read the masked hit register
             -- the input_fifo_enable signal is a read fifo signal
             -- to the centroid's fifo
             -- the hits_busy signal is to indicate that the
             -- hitfilter has started pulling out
             -- the centroids and is busy processing hits
          z_write    : out std_logic; -- fifo enable for the z-axis fifo
         a_centroid_comp  : out unsigned (13 downto 0);
```

```
        -- the extra bit is for the last centroid or end of event bit for the L3
         z_centroid_fifo  : out unsigned (13 downto 0);
         road_event_count : out unsigned (7 downto 0);
             -- this is the internal event counter for the hitfilter
         count            : out unsigned(5 downto 0)
       --this output indicates the number of comparators loaded with road_data
       );
end entity hit_control_block;

--  Architecture Body

architecture behavior of hit_control_block is

type states is (sevent,sfirstload,snext,sload,slast_road,sdecide,
               scentroid_write,scomp_read,shitreg_read,swait,sdata_wait);

signal ncount,pcount,vcount  : unsigned (5 downto 0);
signal nstate, pstate : states;
signal nroad,proad,nmask,pmask: unsigned (5 downto 0);
signal nread_comp,pread_comp: std_logic;
signal nread_hitreg,pread_hitreg: std_logic;
signal ndone,pdone : std_logic;
signal nhits_busy,phits_busy : std_logic;
signal ninput_fifo_enable,pinput_fifo_enable,nfifoempty,pfifoempty,
       nlast_road,plast_road : std_logic;
signal nroad_write,proad_write : std_logic;
signal loadin: std_logic; --signals to load the road_data
signal nroad_event_count,proad_event_count : unsigned (7 downto 0);
signal vroad_event_count : unsigned (7 downto 0);
signal nmux_select,pmux_select : std_logic;
signal mux_select : std_logic;
signal ncentroid, pcentroid : unsigned (12 downto 0);
signal vz_centroid_fifo,va_centroid_comp : unsigned (13 downto 0);
       -- this for temporary assignment of z-centroids and axial centroids
signal nz_write,pz_write : std_logic;
signal nend_of_event, pend_of_event : std_logic;
signal nevent_end,pevent_end : std_logic;
signal nrerr,prerr : std_logic;

constant zero6  : unsigned (5 downto 0)  :="000000";
constant zero13 : unsigned (12 downto 0) := "0000000000000";
constant zero14 : unsigned (13 downto 0) := "00000000000000";
constant zero46 : unsigned (45 downto 0) :=
"0000000000000000000000000000000000000000000000";



begin

ncentroid <= centroid;
nend_of_event <= datatype(2);
 -- the end of event bit is tagged along with the datatype
 -- and the centroids in the input FIFO

-- this process block is used to keep assign a track number
-- to the incoming road data
process (road_write,plast_road,pcount,pstate,pread_comp,done,pdone,pfifoempty,
         phits_busy,pz_write)


begin

-- initialization of the signals
```

```
ncount             <= pcount;
nread_comp         <= pread_comp;
nread_hitreg       <= pread_hitreg;
nstate             <= pstate;
ndone              <= done;
ninput_fifo_enable <= pinput_fifo_enable;
nfifoempty         <= fifoempty;
nroad_write        <= road_write;
nlast_road         <= last_road;
nhits_busy         <= phits_busy;
nroad_event_count  <= proad_event_count;
vroad_event_count  <= "00000000";
nz_write           <= pz_write;
nevent_end         <= pevent_end;


case pstate is

       when sevent =>
              ncount <= zero6;
              vcount <= zero6;
              nhits_busy <= '0';
              loadin <= '0';
              ninput_fifo_enable <= '0';
              nz_write     <= '0';
              nevent_end <= '0';

              if(event_start = '1') then
              -- yes, we have a new event so go to the next state
                     nstate <= sfirstload;
                     nroad_event_count <= proad_event_count + 1;
               -- the new internal event count
              else
                     nstate <= sevent;
              end if;


       when sfirstload =>  -- will load the first road into the comparator

              if (proad_write = '1') then
                     -- do we have a road_write signal only
                     -- then this the first road and there are more to come
                     loadin <= '1';
                     vcount<= zero6;
                     nstate <= snext;
              elsif(plast_road = '1') then
                -- do we have no roads and this is the signal of end of roads
                     nstate <= slast_road;
              else
                     nstate <= sfirstload;
               -- otherwise wait in this state for a road_write signal
              end if;


       when snext =>
              -- checking whether the road_write signal has gone low,
              -- only then can we wait for the next road_write high
              if (proad_write ='0') then
                     nstate <= sload;
                     loadin <= '0';
              else
                     nstate <= snext;
              end if;
```

```
       when sload =>  --loading the next road_data into next comaparator
              if (proad_write = '1') then
                     ncount <= pcount + 1;
                     vcount <= pcount + 1;
                     loadin <= '1';
                     nstate <= snext;
              elsif (plast_road = '1') then
                     nstate <= slast_road;
              else
                     nstate <= sload;
              end if;


       when slast_road =>         -- the previous road was the last raod
                                  -- so now load the centroid
              if (pfifoempty = '0') then
                     ninput_fifo_enable <= '1';
                     nhits_busy <= '1';
                     vroad_event_count <= proad_event_count - 1;
              -- storing the current event value
                     nstate <= sdecide;
              else
                     nstate <= slast_road;
              end if;

       when sdecide =>
              ninput_fifo_enable <= '0';
              if (datatype(1 downto 0) = "10") then -- if the centroid is axial
                     mux_select <= '0';
                     nstate <= scomp_read;
              elsif(datatype(1 downto 0) = "11") then
                        -- if the centroid is z-axis
                     mux_select <= '1';
                     nstate <= scentroid_write;
                     nz_write <= '1';
              else
                     nstate <= slast_road;
              end if;


       when scentroid_write =>
-- this state writes the z-centroid in the block to be formatted
-- and written to the fifo
              nz_write <= '1';
              nstate <= slast_road;

       when scomp_read =>
-- now the comaparator output is read to be read in the
-- hit register to get valid hits
              nread_comp <= '1';  -- active high signal
              nstate <= shitreg_read;

       when shitreg_read =>
              nread_hitreg <= '1';
              nread_comp <= '0';
              nstate <= swait;

       when swait =>
-- wait till the hits for the centroid are found and put in the data format
              nread_hitreg <= '0';
       if (pdone ='1') then
       -- active high signal from the pointer block
              if (pfifoempty = '0'and pend_of_event = '1') then
```

```
	-- indicating it is through with the centroids
			nstate <= sevent;
	-- yes wait for the new event start
			nhits_busy <= '0';
	 -- the hit filter is not busy, the calculation is done
		elsif (pfifoempty = '1' and pend_of_event = '0') then
			-- no not yet done, but fifo is still being loaded
			nstate <= sdata_wait;
		else
			nstate <= slast_road;		-- go pull another centroid
		end if;
	else
		nstate <= swait;
	end if;

	when sdata_wait =>   -- wait for the fifo to go down again
		if (pfifoempty = '0') then
			nstate <= slast_road;
		else
			nstate <= sdata_wait;
		end if;

	when others =>
-- if any other state other than the above defined go to the initial state
		nstate <= sevent;

end case;

end process;


-- this process block is to decide whether axial centroids should go to the
comparator
-- or the zaxis-degree centroids to the fifo
process(pmux_select,pcentroid)

begin
	va_centroid_comp <= zero14;
	vz_centroid_fifo <= zero14;

	case mux_select is

		when '0' =>
			va_centroid_comp <= pend_of_event & pcentroid;

		when '1' =>
			vz_centroid_fifo <= pend_of_event & pcentroid;

		when others =>
			va_centroid_comp <= zero14;
			vz_centroid_fifo <= zero14;

	end case;

end process;


--this block assigns the comparator in which road-data will be loaded


process(loadin,pcount,proad)

variable tcount : unsigned (5 downto 0);
```

```
begin

	nroad <= proad;
	tcount := vcount + 1;

	if(loadin = '1') then   -- active low signal
		road <= tcount;
	else
		road <= zero6;
	end if;

end process;



-- this block give  is out the final mask to be used to determine the hits

process(pread_comp,pmask)

variable tmask : unsigned (5 downto 0);

begin

	tmask := pcount + 1;
	nmask <= pmask;


	if(pread_comp = '1') then
		mask <=tmask;
	else
		mask <= zero6;
	end if;

end process;

-- process block to register the signals

process(clock,reset,nstate,ncount,nread_comp,ndone,nfifoempty,nroad_write,
	nlast_road,nhits_busy,nroad,nmask,nroad_event_count,ncentroid,nevent_end)

begin

	if (reset ='0') then
		pstate             <= sevent;
		pcount             <= zero6;
		pmask              <= zero6;
		proad              <= zero6;
		pread_comp         <= '0';
		pread_hitreg       <= '0';
		pdone              <= '0';
		pfifoempty         <= '0';
		proad_write        <= '0';
		plast_road         <= '0';
		phits_busy         <= '0';
		proad_event_count  <= "00000000";
		pcentroid          <= zero13;
		pevent_end         <= '0';
		pend_of_event      <= '0';
	elsif( clock'event and clock='1') then
		pstate             <= nstate;
		pcount             <= ncount;
		pmask              <= nmask;
		proad              <= nroad;
```

```
              pread_comp         <= nread_comp;
              pread_hitreg       <= nread_hitreg;
              pdone              <= ndone;
              pfifoempty         <= nfifoempty;
              proad_write        <= nroad_write;
              plast_road         <= nlast_road;
              phits_busy         <= nhits_busy;
              proad_event_count  <= nroad_event_count;
              pcentroid          <= ncentroid;
              pevent_end         <= nevent_end;
              pend_of_event      <= nend_of_event;
       end if;

end process;

-- block to register the centroid fifo enable signal
process( clock,reset,ninput_fifo_enable,nz_write)

begin
       if (reset ='0') then
              pinput_fifo_enable <= '0';
              pz_write <= '0';
       elsif( clock'event and clock='0') then
              pinput_fifo_enable <= ninput_fifo_enable;
              pz_write <= nz_write;
       end if;

end process;



-- assignment of all output signals

read_comp         <= pread_comp;
read_hitreg       <= pread_hitreg;
count             <= pcount;
input_fifo_enable <= pinput_fifo_enable;
hits_busy         <= phits_busy;
road_event_count  <= vroad_event_count;
a_centroid_comp   <=  va_centroid_comp;
z_centroid_fifo   <=  vz_centroid_fifo;
z_write           <= pz_write;


end architecture behavior;
```

C.19: Z-CENTROID

```
--------------------------------------------------------------------
--File:z_centroid.vhd
-- To put the z centroids in the right format and then give them to L3
-- and also store them in a FIFO within the design
-- Originally created by : Shweta Lolage : Date 06/29/2000
--------------------------------------------------------------------
-- The initialization of the standard libraries
library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
```

```
-------------------------------------------------------------------------
--  Entity Declaration

entity z_centroid is
   port (clock,reset : in std_logic;
         zvc_enable    : in std_logic;
        -- this signal indicates whether the data should be written to the fifo's
         z_centroid  : in unsigned (13 downto 0);
               -- the z- axis centroid  tagged with the end of event on the MSB
         z_write     : in std_logic;
        -- write signal from the control indicating the centroid coming in
        -- is z axis-type
         l3_trailer  : in unsigned (31 downto 0);
               -- this trailer is passed on from the centroid finder block
         pulse_area  : in unsigned (1 downto 0);
            -- from the centroid finder output FIFO
         z_l3_output : out unsigned (31 downto 0);
                     -- the output which is passed on to the L3 buffer
         z_enable : out std_logic
        );
end entity z_centroid;

--  Architecture Body

architecture behavior of z_centroid is

-- internal signals for the process blocks

signal nz_l3_output,pz_l3_output : unsigned (31 downto 0);
signal nz_enable,pz_enable : std_logic;
signal nl3_trailer,pl3_trailer : unsigned (31 downto 0);
signal pevent_status,nevent_status : std_logic;
signal pcentroid,ncentroid     : unsigned(12 downto 0);
signal ppulse_area, npulse_area  :  unsigned (1 downto 0);

constant c_type     : unsigned (1 downto 0)  := "11";
constant zero2      : unsigned (1 downto 0)  := "00";
constant zero4      : unsigned (3 downto 0)  := "0000";
constant zero5      : unsigned (4 downto 0)  := "00000";
constant zero11     : unsigned (10 downto 0) := "00000000000";
constant zero13     : unsigned (12 downto 0) := "0000000000000";
constant zero14     : unsigned (13 downto 0) := "00000000000000";
constant zero17     : unsigned (16 downto 0) := "00000000000000000";
constant zero32     : unsigned (31 downto 0) :=
"00000000000000000000000000000000";

begin


nl3_trailer   <= l3_trailer;
nevent_status <= z_centroid(13);
ncentroid     <= z_centroid (12 downto 0);
npulse_area   <= pulse_area;

process(zvc_enable,z_write)

begin

nz_l3_output   <= pz_l3_output;


if (zvc_enable = '1' and z_write = '1') then
       nz_enable <= '1';
```

```
       if (pevent_status = '0') then

       nz_l3_output <= "0" & pl3_trailer(21 downto 19) & c_type & ppulse_area &
pl3_trailer(18 downto 8) & pcentroid;

       else
              nz_l3_output <= pl3_trailer;
       end if;
else
       nz_enable <= '0';
end if;


end process;

-- this process block is to register the signals

process( reset,clock,nz_l3_output,nz_enable)

begin

if(reset ='0') then

       pz_l3_output  <= zero32;
       pl3_trailer   <= zero32;
       pcentroid     <= zero13;
       ppulse_area   <= zero2;
       pevent_status <= '0';
       pz_enable <= '0';
elsif (clock'event and clock ='1') then

       pz_l3_output   <= nz_l3_output;
       pl3_trailer    <= nl3_trailer;
       pcentroid      <= ncentroid;
       ppulse_area    <= npulse_area;
       pevent_status  <= nevent_status;
       pz_enable      <= nz_enable;

end if;

end process;

-- assignment of the output signals

       z_l3_output   <= pz_l3_output;
       z_enable      <= pz_enable;

end architecture behavior;
```

C.20: COMPARATOR

```
----------------------------------------------------------------------
--File:comparator.vhd
-- To compare the strip pointer with the data value from the
-- road data
-- Originally created by : Shweta Lolage
--Modified by Shweta Lolage on 05/15/2000 : Changes made according to the new
approach
----------------------------------------------------------------------
```

```
library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;


--  Entity Declaration
entity comparator is
port (clock,reset : in std_logic;
  centroid : in unsigned (10 downto 0);
     -- the centroid value of out of the centroid finder
  uroaddata : in unsigned (10 downto 0); -- the upper road_data address
  lroaddata : in unsigned (10 downto 0); -- the lower road)data address
  loadroad: in std_logic;
    -- load the road data at the inputs into the comparator
  hit : out std_logic      -- to give outthe comparator output
 );
end entity comparator;

--  Architecture Body

architecture behavior of comparator is

-- internal signals for the process blocks
signal ndata,pdata, compare1, compare2 : std_logic;
signal nroaddatau,proaddatau,nroaddatal,proaddatal : unsigned (10 downto 0);
constant zero11 : unsigned ( 10 downto 0) := "00000000000";

begin

-- this process block loads in the upper and lower road-data in the comparator

process(loadroad,uroaddata,lroaddata)

begin

if (loadroad = '1') then
       nroaddatal <= lroaddata;
       nroaddatau <= uroaddata;
else
       nroaddatal <= proaddatal;
       nroaddatau <= proaddatau;
end if;

end process;


-- this block is used for the comparison of the road-data
-- with the given centroid

process(proaddatau,proaddatal,centroid)

begin
       compare1 <= '0';
       compare2 <= '0';

       if( proaddatau /= zero11 and centroid /= zero11 ) then
              if (proaddatau >= centroid ) then
                     compare1 <= '1';
              else
                     compare1 <= '0';
```

```
		end if;
	else
		compare1 <= '0';
	end if;


	if(proaddatal /= zero11 and centroid /= zero11) then
		if (proaddatal <= centroid) then
			compare2 <= '1';
		else
			compare2 <= '0';
		end if;
	else
		compare2 <= '0';
	end if;

end process;

-- this block gives the result of the comparison
process (compare1,compare2)

begin
	ndata <= pdata;
	if (compare1 ='1' and compare2 ='1') then
		ndata <= '1';
	else
		ndata <= '0';
	end if;

end process;


-- the process block to register the road-data signals

process( nroaddatau,nroaddatal,ndata,clock,reset)

begin
	if ( reset ='0') then
		proaddatau <= zero11;
		proaddatal <= zero11;
		pdata <= '0';
	elsif ( clock'event and clock='1') then
		proaddatau <= nroaddatau;
		proaddatal <= nroaddatal;
		pdata <= ndata;
	end if;

end process;


-- final out from the comparator
hit <= pdata;

end architecture behavior;
```

C.21: HIT REGISTER

```
----------------------------------------------------------------------
-- File:r2_hitreg_v2.vhd
```

```
-- this process block finds the valid hits
-- Originally created by : Shweta Lolage
-- By Shweta Lolage on 05/17/2000 : Changes made according to the new approach
--modified on  06/04 /2000 : by Shweta Lolage
-- modified to remove the delays
-- The control output line removed from this block and put in the control logic
as an additional state
-- date - 09/04/2000 - Shweta Lolage
----------------------------------------------------------------------

library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;


--  Entity Declaration
entity hitreg is
port (clock, reset : in std_logic;
      read_comp : in std_logic;  -- read signal from the control block
      mask,hitreg : in unsigned (45 downto 0);
        -- the mask register and the hit register used to find the valid hits
      hitbit : out std_logic_vector (45 downto 0) -- valid hits
     );
end entity hitreg;

--  Architecture Body

architecture behavior of hitreg is

signal nhitbit, phitbit : std_logic_vector (45 downto 0);
constant zero46 : std_logic_vector (45 downto 0) :=
"0000000000000000000000000000000000000000000000";

begin

-- process block to register in the mask and the hits register

process (mask,hitreg)

begin

if (read_comp = '1') then

       nhitbit(0)  <= mask(0)  and hitreg(0);
       nhitbit(1)  <= mask(1)  and hitreg(1);
       nhitbit(2)  <= mask(2)  and hitreg(2);
       nhitbit(3)  <= mask(3)  and hitreg(3);
       nhitbit(4)  <= mask(4)  and hitreg(4);
       nhitbit(5)  <= mask(5)  and hitreg(5);
       nhitbit(6)  <= mask(6)  and hitreg(6);
       nhitbit(7)  <= mask(7)  and hitreg(7);
       nhitbit(8)  <= mask(8)  and hitreg(8);
       nhitbit(9)  <= mask(9)  and hitreg(9);
       nhitbit(10) <= mask(10) and hitreg(10);
       nhitbit(11) <= mask(11) and hitreg(11);
       nhitbit(12) <= mask(12) and hitreg(12);
       nhitbit(13) <= mask(13) and hitreg(13);
       nhitbit(14) <= mask(14) and hitreg(14);
       nhitbit(15) <= mask(15) and hitreg(15);
       nhitbit(16) <= mask(16) and hitreg(16);
       nhitbit(17) <= mask(17) and hitreg(17);
```

```
       nhitbit(18) <= mask(18) and hitreg(18);
       nhitbit(19) <= mask(19) and hitreg(19);
       nhitbit(20) <= mask(20) and hitreg(20);
       nhitbit(21) <= mask(21) and hitreg(21);
       nhitbit(22) <= mask(22) and hitreg(22);
       nhitbit(23) <= mask(23) and hitreg(23);
       nhitbit(24) <= mask(24) and hitreg(24);
       nhitbit(25) <= mask(25) and hitreg(25);
       nhitbit(26) <= mask(26) and hitreg(26);
       nhitbit(27) <= mask(27) and hitreg(27);
       nhitbit(28) <= mask(28) and hitreg(28);
       nhitbit(29) <= mask(29) and hitreg(29);
       nhitbit(30) <= mask(30) and hitreg(30);
       nhitbit(31) <= mask(31) and hitreg(31);
       nhitbit(32) <= mask(32) and hitreg(32);
       nhitbit(33) <= mask(33) and hitreg(33);
       nhitbit(34) <= mask(34) and hitreg(34);
       nhitbit(35) <= mask(35) and hitreg(35);
       nhitbit(36) <= mask(36) and hitreg(36);
       nhitbit(37) <= mask(37) and hitreg(37);
       nhitbit(38) <= mask(38) and hitreg(38);
       nhitbit(39) <= mask(39) and hitreg(39);
       nhitbit(40) <= mask(40) and hitreg(40);
       nhitbit(41) <= mask(41) and hitreg(41);
       nhitbit(42) <= mask(42) and hitreg(42);
       nhitbit(43) <= mask(43) and hitreg(43);
       nhitbit(44) <= mask(44) and hitreg(44);
       nhitbit(45) <= mask(45) and hitreg(45);
else
       nhitbit <= zero46;

end if;

end process;


-- process block to register the signals

process(clock,reset,nhitbit)

begin

if ( reset = '0') then
       phitbit <= zero46;
elsif ( clock'event and clock ='1') then
       phitbit <= nhitbit;
end if;

end process;

-- assigning the outputs
hitbit <= phitbit;

end architecture;
```

C.22: HIT FORMAT

```
----------------------------------------------------------------------
-- File:hit_format.vhd
-- Process block to locate the hits from the valid hits register
```

```
-- and give them to the format block to put them in the data format
-- Originally created by : Shweta Lolage
-- By Shweta Lolage on 05/17/2000 : Changes made according to the new approach
-- Changes are made to the design to combine the hit block and the format block
-- date : 09/04/2000 - Shweta Lolage
----------------------------------------------------------------------

library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;


--  Entity Declaration
entity hit_format is
       port (clock,reset : in std_logic;
         count : in unsigned (5 downto 0);
-- upper limit for the hit search as this is the number of comparators
-- loaded with valid road_data
         hitreg: in std_logic_vector (45 downto 0);
              -- valid hits register
         l3_trailer : in unsigned (31 downto 0) ;
                    -- this trailer is passed on from the centroid finder block
         pulsearea : in unsigned (1 downto 0); -- pulse_area for the centroid
         centroid : in unsigned (13 downto 0); -- centroid which is compared
         hitreg_valid : in std_logic;          -- read valid hit register signal
         cen_done,hits_done : out std_logic;
                    --cen_done signal for the channel control block
        output : out unsigned (32 downto 0);
              -- the final output to the hits FIFO

         fifo_enable,end_of_event  : out std_logic
                    -- signal to the main control block indicating that the --
                    -- channel has valid hits
       );
end entity hit_format;

--  Architecture Body

architecture behavior of hit_format is

type hitstates is (sinit,spoint_select,sready,sread_hits,sread_bit,swrite_hit,
                   soutput_hit,snext_hit,strailer);

signal pstate,nstate : hitstates;
signal or1,or2,or3,or4,or5 : std_logic;
signal nor1,nor2,nor3,nor4,nor5 : std_logic;
signal por1,por2,por3,por4,por5 : std_logic;
signal nhits_present,phits_present,pcen_done,ncen_done : std_logic;
signal npointer,ppointer: unsigned (5 downto 0);
signal ncounter,pcounter: unsigned (5 downto 0);
signal nhitreg_valid,phitreg_valid : std_logic;
signal nhitreg, phitreg : std_logic_vector(45 downto 0);
signal nhits_done,phits_done,nbit,pbit : std_logic;

constant zero46 : std_logic_vector (45 downto 0) :=
"0000000000000000000000000000000000000000000000";


signal ndata,pdata: unsigned (32 downto 0);
signal ncount,pcount : unsigned (5 downto 0);
signal noutput,poutput : unsigned (32 downto 0);
```

```
signal nfifoenable,pfifoenable : std_logic;
signal npulsearea,ppulsearea : unsigned (1 downto 0);
signal pcentroid, ncentroid : unsigned (12 downto 0);
signal nevent_status, pevent_status : std_logic;
signal nl3_trailer,pl3_trailer : unsigned (31 downto 0);

constant zero33 : unsigned (32 downto 0) :=
"0000000000000000000000000000000000";
constant zero32 : unsigned (31 downto 0) := "00000000000000000000000000000000";
constant zero22 : unsigned (21 downto 0) := "0000000000000000000000";
constant zero17 : unsigned (16 downto 0) := "00000000000000000";
constant zero16 : unsigned (15 downto 0) := "0000000000000000";
constant zero11 : unsigned (10 downto 0) := "00000000000";
constant zero8  : unsigned (7 downto 0)  := "00000000";
constant zero6  : unsigned (5 downto 0)  := "000000";
constant zero3  : unsigned (2 downto 0)  := "000";
constant two    : unsigned (1 downto 0)  := "10";
constant three  : unsigned (1 downto 0)  := "11";

begin

-- the initialization of the signals
       npulsearea <= pulsearea;
       ncentroid <= centroid(12 downto 0);
       nevent_status <= centroid(13);
-- the bit tagged along with centroid indicating the end of event
       nl3_trailer <= l3_trailer;

       nhitreg_valid <= hitreg_valid;

-- this process block groups the comparators to reduce the search
process(hitreg)

variable tor1,tor2,tor3,tor4,tor5 : std_logic;

begin
       tor1 := '0';
       tor2 := '0';
       tor3 := '0';
       tor4 := '0';
       tor5 := '0';


for i in 0 to 9 loop
       tor1 := tor1 or hitreg(i);
end loop;

       or1 <= tor1;


for i in 10 to 19 loop
       tor2 := tor2 or hitreg(i);
end loop;

       or2 <= tor2;

for i in 20 to 29 loop
       tor3 := tor3 or hitreg(i);
end loop;

       or3 <= tor3;

for i in 30 to 39 loop
```

```
       tor4 := tor4 or hitreg(i);
end loop;

       or4 <= tor4;

for i in 40 to 45 loop
       tor5 := tor5 or hitreg(i);
end loop;

       or5 <= tor5;


end process;



-- this process block is to find the hits and send them to the format block
process(ppointer,hitreg_valid,count,pcen_done,pcen_done,pcounter)

begin

       ncen_done <= pcen_done;
       ncounter  <= pcounter;

       nor1     <= por1;
       nor2     <= por2;
       nor3     <= por3;
       nor4     <= por4;
       nor5     <= por5;

       nhits_present <= phits_present;
       npointer <= ppointer;
       nhitreg <= phitreg;
       nfifoenable <= pfifoenable;
       noutput <= poutput;
       nhits_done <= phits_done;




case pstate is

       when sinit =>       -- initial state,
              ncen_done <= '0';
              nhits_present <= '0';
              nfifoenable <= '0';
              nhits_done <= '0';
       if (hitreg_valid ='1') then      -- is valid register ready to be read
              nstate <= spoint_select;
                     -- yes, go ahead and select the point of start
                     -- after determining whether we have any hits
              nhitreg <= hitreg;
              nor1  <= or1;
              nor2  <= or2;
              nor3  <= or3;
              nor4  <= or4;
              nor5  <= or5;
       else
              nstate <= sinit;
            -- no be in this state and wait for it
              nhitreg <= zero46;
              nor1  <='0';
              nor2  <='0';
```

```
              nor3  <='0';
              nor4  <='0';
              nor5  <='0';
       end if;

       when spoint_select =>

              nfifoenable <= '0';
              nhits_present <= por1 or por2 or por3 or por4 or por5;

              if (por1 ='1') then
                     npointer <= "000000";
              else
              if (por2 ='1') then
               npointer <= "001010";
              else
               if (por3 ='1') then
                npointer <= "010100";
               else
                 if (por4='1') then
                   npointer <= "011110";
                 else
                   if (por5 = '1') then
                     npointer <= "101000";
                  else
                     npointer <= "000000";
                  end if;
              end if;
       end if;
   end if;
end if;

       nstate <= sready;



when sready =>   -- read the valid hits
       nfifoenable <= '0';

       if (phits_present = '1') then  -- are there any hits in this register
              nstate <= sread_hits;            -- yes there are
              ncen_done <= '0';
              ncounter <= ppointer;
       else
              ncen_done <= '1';
              nstate <= sinit;
       end if;

when sread_hits =>

       nstate     <= sread_bit;
       nfifoenable <= '0';


       case pcounter is

        when "000000" => nbit <= phitreg(0);
        when "000001" => nbit <= phitreg(1);
        when "000010" => nbit <= phitreg(2);
        when "000011" => nbit <= phitreg(3);
        when "000100" => nbit <= phitreg(4);
         when "000101" => nbit <= phitreg(5);
        when "000110" => nbit <= phitreg(6);
```

```
         when "000111" => nbit <= phitreg(7);
         when "001000" => nbit <= phitreg(8);
         when "001001" => nbit <= phitreg(9);
         when "001010" => nbit <= phitreg(10);
         when "001011" => nbit <= phitreg(11);
         when "001100" => nbit <= phitreg(12);
         when "001101" => nbit <= phitreg(13);
         when "001110" => nbit <= phitreg(14);
         when "001111" => nbit <= phitreg(15);
         when "010000" => nbit <= phitreg(16);
         when "010001" => nbit <= phitreg(17);
         when "010010" => nbit <= phitreg(18);
         when "010011" => nbit <= phitreg(19);
         when "010100" => nbit <= phitreg(20);
         when "010101" => nbit <= phitreg(21);
         when "010110" => nbit <= phitreg(22);
         when "010111" => nbit <= phitreg(23);
         when "011000" => nbit <= phitreg(24);
         when "011001" => nbit <= phitreg(25);
         when "011010" => nbit <= phitreg(26);
         when "011011" => nbit <= phitreg(27);
         when "011100" => nbit <= phitreg(28);
         when "011101" => nbit <= phitreg(29);
         when "011110" => nbit <= phitreg(30);
         when "011111" => nbit <= phitreg(31);
         when "100000" => nbit <= phitreg(32);
         when "100001" => nbit <= phitreg(33);
         when "100010" => nbit <= phitreg(34);
         when "100011" => nbit <= phitreg(35);
         when "100100" => nbit <= phitreg(36);
         when "100101" => nbit <= phitreg(37);
         when "100110" => nbit <= phitreg(38);
         when "100111" => nbit <= phitreg(39);
         when "101000" => nbit <= phitreg(40);
         when "101001" => nbit <= phitreg(41);
         when "101010" => nbit <= phitreg(42);
         when "101011" => nbit <= phitreg(43);
         when "101100" => nbit <= phitreg(44);
         when "101101" => nbit <= phitreg(45);
         when others   => nbit <= '0';

end case;

when sread_bit =>

       nfifoenable <= '0';
       if (pbit = '1') then  -- no there are more
              nstate <= swrite_hit;
              ncount <= pcounter;
       else
              nstate <= snext_hit;
              ncount <= pcount;
       end if;

when swrite_hit =>
       nstate <= soutput_hit;
       nfifoenable <= '0';
       ndata(32)           <= '0';
       ndata(31 downto 26) <= pcount;
       ndata(25 downto 24) <= ppulsearea;
       ndata(23 downto 13) <= pl3_trailer(18 downto 8);
       ndata(12 downto 0)  <= pcentroid;
```

```
when soutput_hit =>
       nstate <= snext_hit;
       noutput<= pdata;
       nfifoenable <= '1';

when snext_hit =>
-- state to decide whether we should go ahead for the next hit

       nfifoenable <= '0';
       if (pcounter <= count) then
                    -- have we reached the end of the comparators
             ncen_done <='0';
             ncounter <= pcounter + 1;         -- No , the end has not yet come
              nstate <= sread_hits;
        else
             ncen_done  <= '1';
             ncounter <= "000000";
             if (pevent_status = '0')then
                    nstate <= sinit;
                -- yes go to start and wait for the next lot
             else
                    nstate <= strailer;
                    nhits_done <= '1';
                    ndata (32)          <= '1';
                    ndata(31 downto 0)  <= pl3_trailer;
             end if;
         end if;

when strailer =>
       nstate <= sinit;
       nfifoenable <= '1';
       noutput <= pdata;
       nhits_done <= '0';

when others =>
-- any other state other than the once defined above, go to the initial state
        nstate <= sinit;
        ncen_done <= '0';
        nhits_done <= '0';

end case;
end process;

--this process block registers the signals

process(clock,reset)

begin

       if (reset ='0') then


             ppointer  <="000000";
             pcounter <="000000";
             phits_present <= '0';
             pstate <= sinit;
             pcen_done <= '0';
             pbit <= '0';
             phitreg_valid <= '0';
             por1  <= '0';
             por2  <= '0';
             por3  <= '0';
             por4  <= '0';
```

```
              por5  <= '0';
              phitreg <= zero46;

       elsif (clock'event and clock ='1') then


              ppointer      <= npointer;
              pcounter      <= ncounter;
              phits_present <= nhits_present;
              pstate        <= nstate;
              pcen_done     <= ncen_done;
              pbit          <= nbit;
              phitreg_valid <= nhitreg_valid;
              por1          <= nor1;
              por2                  <= nor2;
              por3            <= nor3;
              por4         <= nor4;
              por5            <= nor5;
              phitreg       <= nhitreg;
       end if;

end process;


-- the process block to register the signals
process(clock,reset)

begin

if(reset ='0') then
       poutput <= zero33;
       pcount <= zero6;
       pdata <= zero33;
       ppulsearea <= "00";
       pcentroid <= "0000000000000";
       pevent_status <= '0';
       pl3_trailer <= zero32;
       pfifoenable <= '0';
       phits_done <= '0';
elsif( clock'event and clock='1') then
       poutput <= noutput;
       pcount <= ncount;
       pdata <= ndata;
       ppulsearea <= npulsearea;
       pcentroid <= ncentroid;
       pl3_trailer <= nl3_trailer;
       pevent_status <= nevent_status;
       pfifoenable <= nfifoenable;
       phits_done <= nhits_done;
end if;

end process;

-- final out to the fifo, this block is not totally complete as will have to
see the handshaking with the fifo
       output <= poutput;
       fifo_enable <= pfifoenable;
       end_of_event <= poutput(32);
       cen_done  <= pcen_done;
       hits_done <= phits_done;

end architecture;
```

C.23: HIT READ OUT

```
-------------------------------------------------------------------
-- File:hit_readout.vhd
-- Process block to read the hits from the FIFO by the control block
-- Originally created by : Shweta Lolage :08/01/2000
-------------------------------------------------------------------

library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;


--  Entity Declaration
entity hit_readout is
       port (clock, reset : in std_logic;
                hits_enable,hit_fifo_empty, hits_busy : in std_logic;
                read_hit_fifo,hits_available : out std_logic
               );
end entity hit_readout;

--  Architecture Body

architecture behavior of hit_readout is

signal nread_hit_fifo,pread_hit_fifo : std_logic;

begin

       process( hits_enable, hit_fifo_empty)

       begin


       if(hits_enable = '1' and hit_fifo_empty = '0') then
              nread_hit_fifo <= '1';  -- this is  read_req signal to the fifo
       else
              nread_hit_fifo <= '0';
       end if;

end process;

hits_available <=  hits_busy;
-- this is an internal signal to the read_out control to indicate that hits are
being processed


reg : process ( clock, reset)

       begin

       if(reset = '0') then
              pread_hit_fifo <= '0';
       elsif (clock'event and clock ='1') then
              pread_hit_fifo <= nread_hit_fifo;
       end if;

end process reg;
```

```
-- assigning the outputs
read_hit_fifo <= pread_hit_fifo;


end architecture;
```

C.24: TRAILER PATTERN RECOGINSER

```
----------------------------------------------------------------------
-- File:trailer_pat.vhd
-- Process block to check for the trailer
-- Originally created by : Shweta Lolage :08/01/2000
----------------------------------------------------------------------

library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;


--  Entity Declaration
entity trailer_pat is
       port (stream : in unsigned(4 downto 0);
                trailer_signal : out std_logic
              );
end entity trailer_pat;

--  Architecture Body

architecture behavior of trailer_pat is

constant pattern : unsigned (4 downto 0):= "11110";

begin

       with stream select
              trailer_signal <= '1' when pattern,
                                '0' when others;

end architecture;
```

C.25 DETAILED MEMORY ALLOCATION OF THE DOWNLOADED PARAMETERS

1. Monitor space: 1K memory: 0000 – 03FF.
2. Miscellaneous: 1K memory: 0400 – 07FF.
3. Gain Offset LUT: 4K X 8: 0800 – 17FF
4. Test data LUT: 1K (default) X 18: 1800 – 1BFF.
5. Empty Space: 1C00 – 3FFF.
6. Road data LUT: 16 K X 22: 4000 – 7FFF.

C.25.1 The monitoring data memory allocation

1. SERR,mismatch,SMT_ID,SEQ_ID,HDI_ID,data (last data word read): address 0000.
2. SERR error count : address 0001.
3. Mismatch counter: mismatch in the SEQ-ID or the HDI-ID: address 0002.
4. Zero error count: Zero byte not found: address 0003.
5. Number of undefined data_type clusters: address 0010.
6. Number of axial data_type clusters: address 0011.
7. Number of stereo data_type clusters: address 0012.
8. Number of Z-axis data_type clusters: address 0013.
9. Data activity on Chip(I): address 0020-0028.

C.25.2 The miscellaneous memory space memory allocation

1. Bad channel: 32 X 64: address: 0400 – 047F.
2. Chip ranges: 24 X 1: address: 0480.
3. Pulse area thresholds: 24 X 4 X4: address: 0500 – 0503.
4. Miscellaneous parameters: 32 X 1: address: 0580.
5. Cluster threshold values: 17 X 4: address: 0600 – 0603.