Implementing Multi-Port Memories in Axcelerator Devices

Introduction

This application note describes a user configurable VHDL wrapper for implementing dual-port and quad-port memory structures using a small number of programmable logic tiles and the embedded memory blocks in Actel's Axcelerator Field Programmable Gate Array (FPGA) devices.

The Axcelerator device architecture provides dedicated blocks of RAM with independent read and write ports, which are completely independent and fully synchronous. Each memory block consists of 4,608 bits with independent read and write port configuration, which can be organized as 128x36, 256x18, 512x9, 1kx4, 2kx2, or 4kx1 to allow for built-in bus width conversion and can be cascaded to create larger memory sizes. For additional details on embedded memory blocks in Axcelerator devices, refer to the *Axcelerator Family FPGAs* datasheet or the application note *Axcelerator Family Memory Blocks*. A block diagram of the basic memory block is shown in Figure 1.

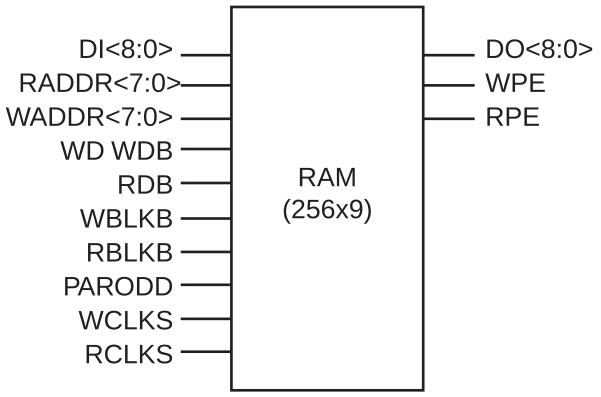

Figure 1* • *Basic Axcelerator SRAM Module

The memory blocks in the Axcelerator devices can be used to implement multi-port memories with the addition of some simple multiplex logic and an extra clock operating at double the read and write clock frequency.

Basics of Multi-Port Memories

This application note discusses two types of multi-port memories: dual-port and quad-port. In both configurations, two data access ports (data port A and data port B) can be used to perform read and write operations into the Axcelerator RAMs. Each data port has its own data bus, address bus, read enable, and write enable signals. The basic principle of implementing multi-port memories in Axcelerator devices involves the use of an additional clock operating at double the read and write clock frequency to access the memory space through some multiplex logic and arbitrate between data access ports. The overall bandwidth of the memory (bit/s) remains the same, and the only difference between the single and multi-port memory is read/write frequency versus data-width trade-off.

Although more than one data access port is now available, they share the same memory space. Since Axcelerator memories are synchronized, simultaneous read/write cycles to the same memory address during a single data cycle will result in data being correctly written, but it is unclear which value the read port will output.

Dual-Port Memory

The dual-port memory configuration consists of two data access ports (two read/write ports) sharing a single clock domain (wr_clk). The write address bus from each data access port (a_wadr and b_wadr) is used for both read and write operations. The read enable (a_rdblk, a_rdb, b_rdblk, and b_rdb) and write enable (a_wrblk, a_wrb, b_wrblk, and b_wrb) signals are used to select between either read or write operation for each data access port. Figure 2 on page 2 shows the corresponding ports of a dual-port memory block.

Quad-Port Memory

The quad-port memory configuration consists of two data access ports, each with a separate write port and read port, clocked by separate write (wr_clk) and read (rd_clk) clocks. For each data access port, there are separate address busses used to perform read (a_radr and b_radr) and write (a_wadr and b_wadr) operations. The read enable (a_rdblk, a_rdb, b_rdblk, and b_rdb) and write enable (a_wrblk, a_wrb, b_wrblk, and b_wrb) signals are used to activate the read and write operations for each data access port. Figure 3 on page 2 shows the corresponding ports of a quad-port memory block.

Table 1 on page 3 summarizes the interface signals of the memory block.

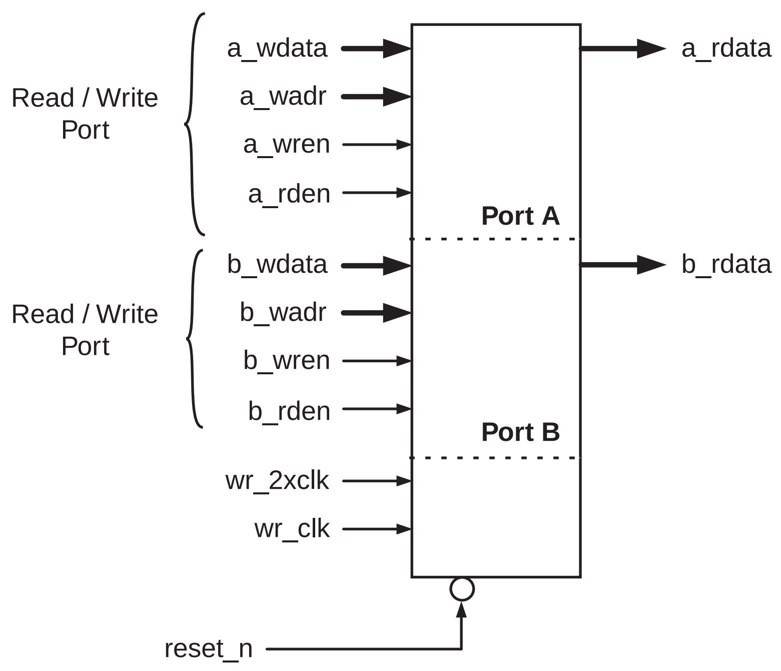

***Figure 2* • Dual-Port Memory Block Interface Signals**

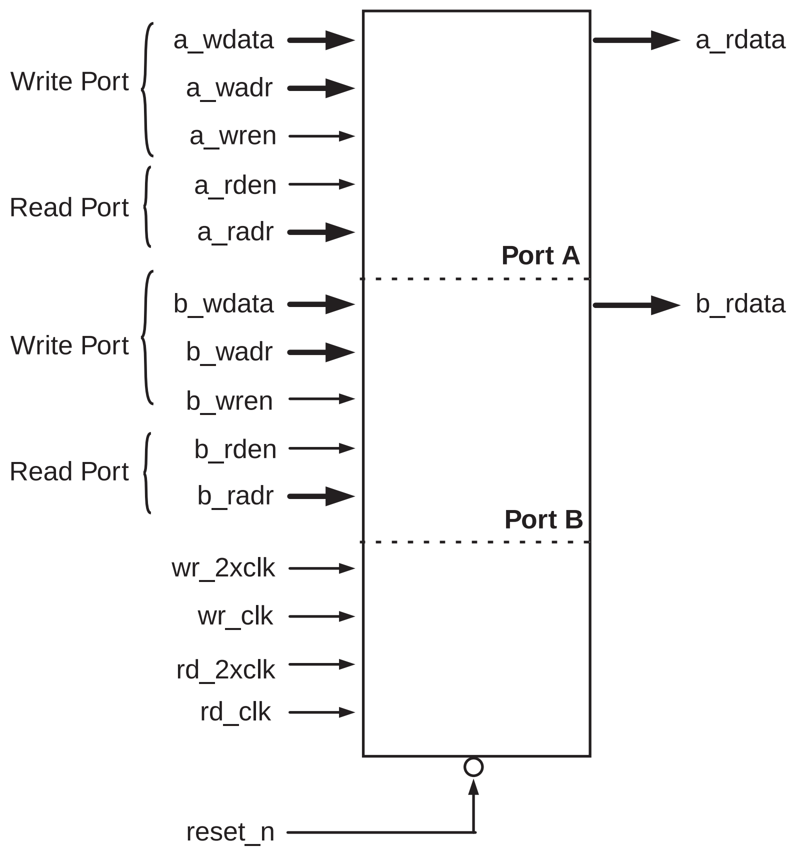

***Figure 3* • Four-Port Memory Block Interface Signals**

Table 1 **•** *Multi-Port Memory Interface Signals*

Signal	**Bits**	**In/Out**	**Description**
a_wdata	variable	IN	Write data bus
a_wadr	8	IN	Write / dual-port memory address
a_wren	1	IN	Active high data enable
a_rdata	variable	IN	Output data bus
a_radr	8	OUT	Output address bus (quad-port memory mode only)
a_rden	1	IN	Output register enable A
b_wdata	variable	IN	Write data bus
b_wadr	8	IN	Write / dual-port memory address bus
b_wren	1	IN	Active high data enable
b_rdata	variable	OUT	Output data bus
b_radr	8	IN	Output address bus (quad-port memory mode only)
b_rden	1	IN	Output register enable B
wr_2xclk	1	IN	2x write clock
wr_clk	1	IN	Write port data clock / multiplexer select
rd_2xclk	1	IN	2x read clock (quad-port memory mode only)
rd_clk	1	IN	Read data clock (quad-port memory mode only)
reset_n	1	IN	Reset signal (active low)

Implementing Multi-Port Memories

In the referenced example (see the "Appendix – Design Example" on page 8), the multi-port memory wrapper can be implemented in two configurations as described above: dual-port memory (PMODE=0) and quad-port memory (PMODE=1). The depth of the implemented multi-port memories is limited to a single memory block, but the width is variable and supports the variable aspect ratio for quad-port memory configuration. Axcelerator's RAM64K36 macro is used as the basic memory block for this wrapper.

Since the implementation of multi-port memories rely on the fact that the embedded memory block is clocked at twice the data clock rate, a double frequency clock must be generated. The original data clock input is easily doubled in frequency to generate the required wr_2xclk and rd_2xclk signals using the PLLs in the Axcelerator architecture. For additional details on how to generate a PLL for Axcelerator devices, please refer to *A Guide to ACTgen Macros* or the application note *Axcelerator Family PLL and Clock Management*.

Table 2 lists the configurable parameters for the reference design in the "Appendix – Design Example" on page 8, which will be explained in the following sections.

Table 2 **•** *Configurable Parameters for Design Example in Appendix*

Parameter	**Value**	**Description**
PMODE	0 (default)	Dual-port memory configuration
	1	Quad-port memory configuration
WMODE	0 (default)	No register, only MUX logic for waddr
	1	Register, then multiplex waddr, wenb, and wdata
RMODE	0 (default)	No register, only MUX logic for raddr
	1	Register, then multiplex raddr
OREG	0 (default)	Transparent output mode
	1	Registered output mode
WR_DEPTH	128:4096 (default=256)	Write port depth
WR_WIDTH	1:288 (default = 36)	Write port width
WR_DEPTH	128:4096 (default = 512)	Read port depth
WR_WIDTH	calculated	Read port width

Read Ports - Dual-Port Memory

Figure 4 shows a block diagram of the dual-port memory implementation.

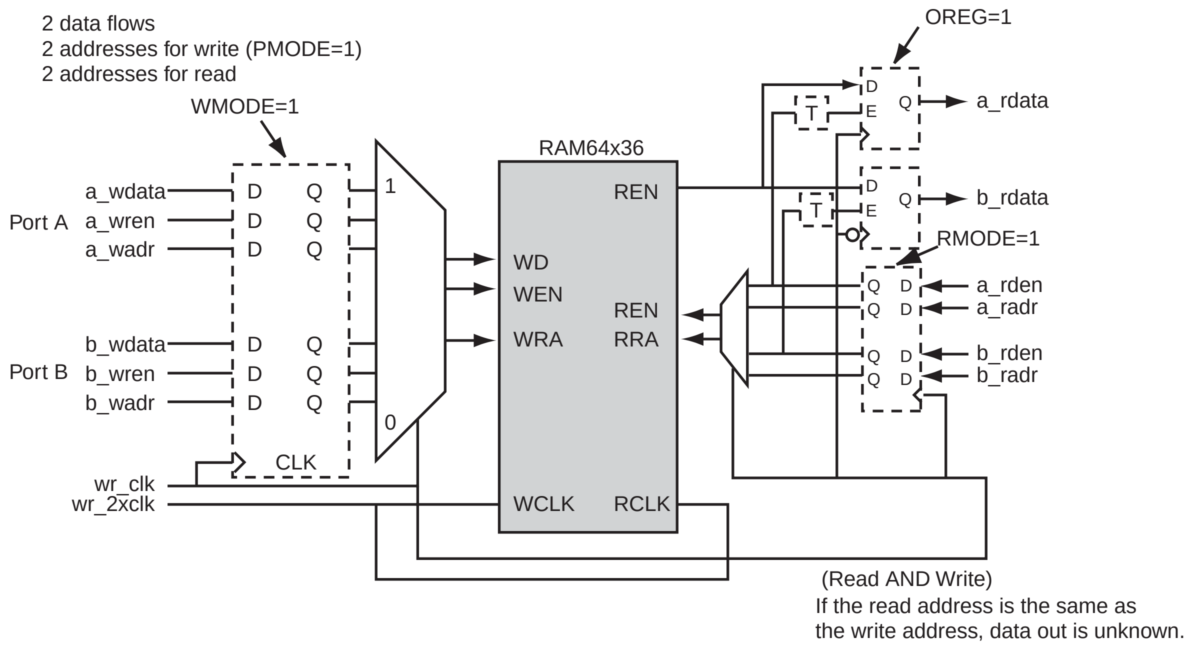

Figure 4 **•** *Dual-Port Memory Implementation*

In this configuration, the write addresses and write clock are used to read the memory, while the read address inputs and read clock remain unused in the code. The values for read port depth (parameter: RD_DEPTH) and write port depth (parameter: WR_DEPTH) have no specific required relationship, and can be: 128, 256, 512, 1,024, 2,048, and 4,096 bits.

If OREG = ’0,’ the data outputs propagate directly to the data output ports; otherwise, when OREG = ’1,’ the output is valid with the next rising-edge of wr_clk. Data for Read/Write Port A is registered on the falling-edge of wr_clk, while data for Read/Write Port B is registered on the rising-edge of wr_clk. The read operation supported by this wrapper is synchronous. The read enable inputs are used as inputs to enable the output registers.

Read Ports – Quad-Port Memory

Figure 5 on page 5 shows a block diagram of the quad-port memory implementation.

If RMODE is set to ’1,’ the address and enables for both read ports are registered with the rising edge of the read clock rd_clk. Then the read address and enable inputs for Port A and Port B are multiplexed to the memory and settle while rd_clk signal is high and low respectively. Data is read from the memory on the next rising edge of rd_2xclk. If RMODE is set to ’0,’ the read address and enables are not registered and the read address and read enable inputs for Port A and Port B are simply multiplexed to access the Axcelerator memory.

If OREG = ’0,’ the data outputs propagate directly to the data output ports; otherwise, when OREG = ’1,’ the output is valid with the rising-edge of rd_clk. Read data for Port A is registered on the falling-edge of rd_clk, while read data for Port B is registered on the rising-edge of rd_clk. The read operation supported by this wrapper is synchronous. The read enable inputs are used to enable the output registers.

Write Ports

The Write Port implementation for both dual-port and quad-port memory is the same. Data, address, and enables for both write ports are optionally registered with the rising edge of wr_clk when WMODE = 1. Data, address, and enable signals for Port A and Port B are multiplexed to the memory and settle while wr_clk signal is high and low respectively. Then data is written into the memory on the next rising edge of wr_2xclk (next falling or rising edge of wr_clk).

Width / Depth

Because the embedded memories in the Axcelerator family have variable aspect ratio, this wrapper supports independent width and depth for write and read ports in the quad-port memory configuration. The user specifies the depth and width for the write ports, and depth only for the read ports – read port width is derived from the other values in the wrapper code.

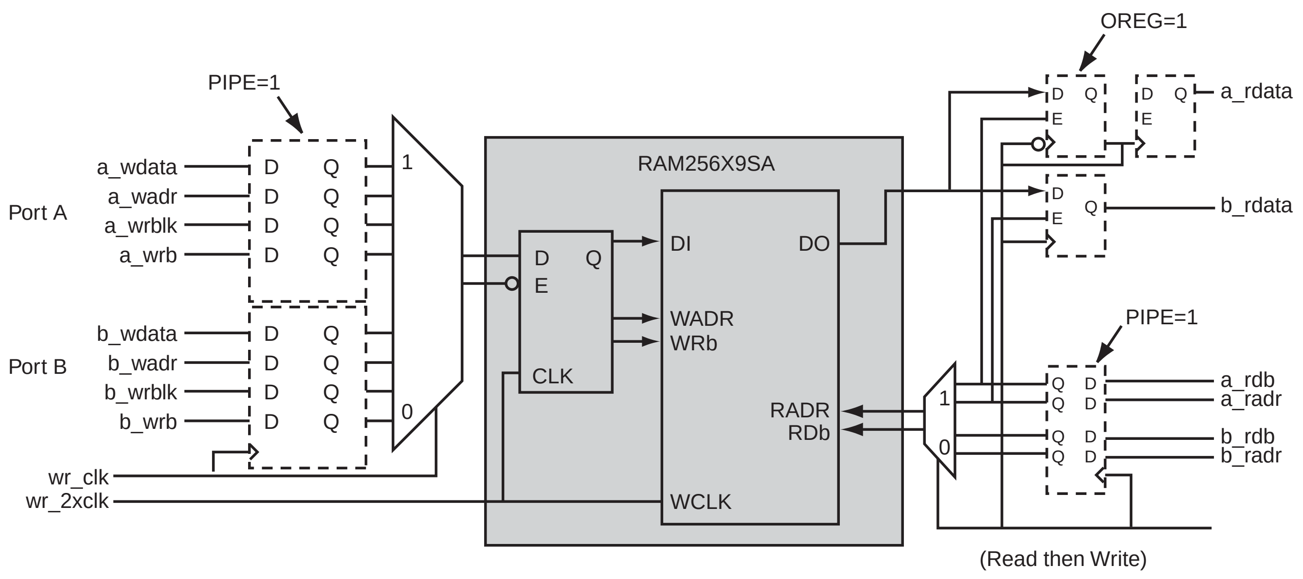

Figure 5 • *Quad-Port Memory Implementation*

Write data width is specified in integer multiples of the word-width associated with the specified write depth (refer to the *Axcelerator Family FPGAs* datasheet*)*. For example, if the specified write depth is 256, the width may be any multiple of 18. Care must be taken to specify correct width and depth parameters, as the user specified GENERIC values are not validated by the wrapper.

Timing Diagrams

Figure 6 and Figure 7 on page 6 illustrate the relationships of the signals during Write and Read Cycles for both dual-port and quad-port memories.

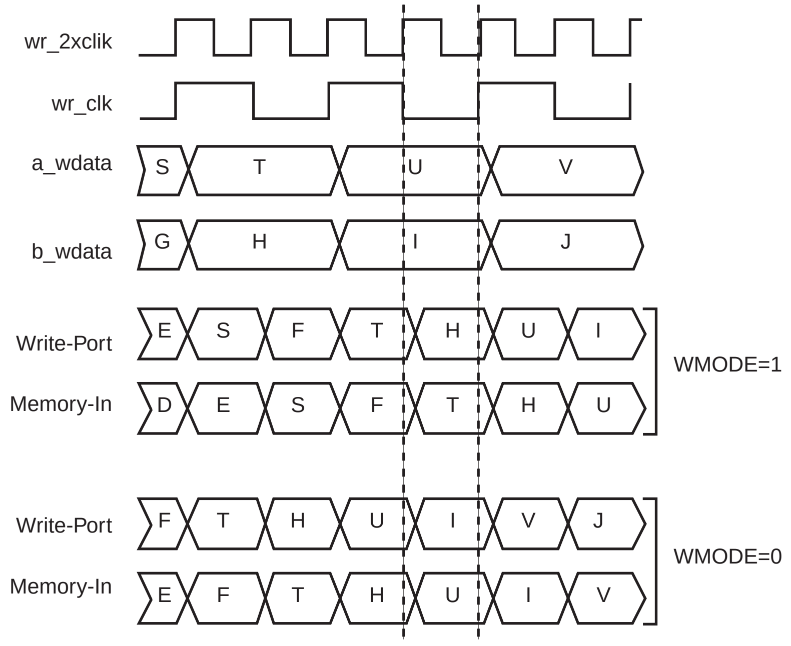

Figure 6 • *Axcelerator Multi-Port Memory Implementation Write Cycle*

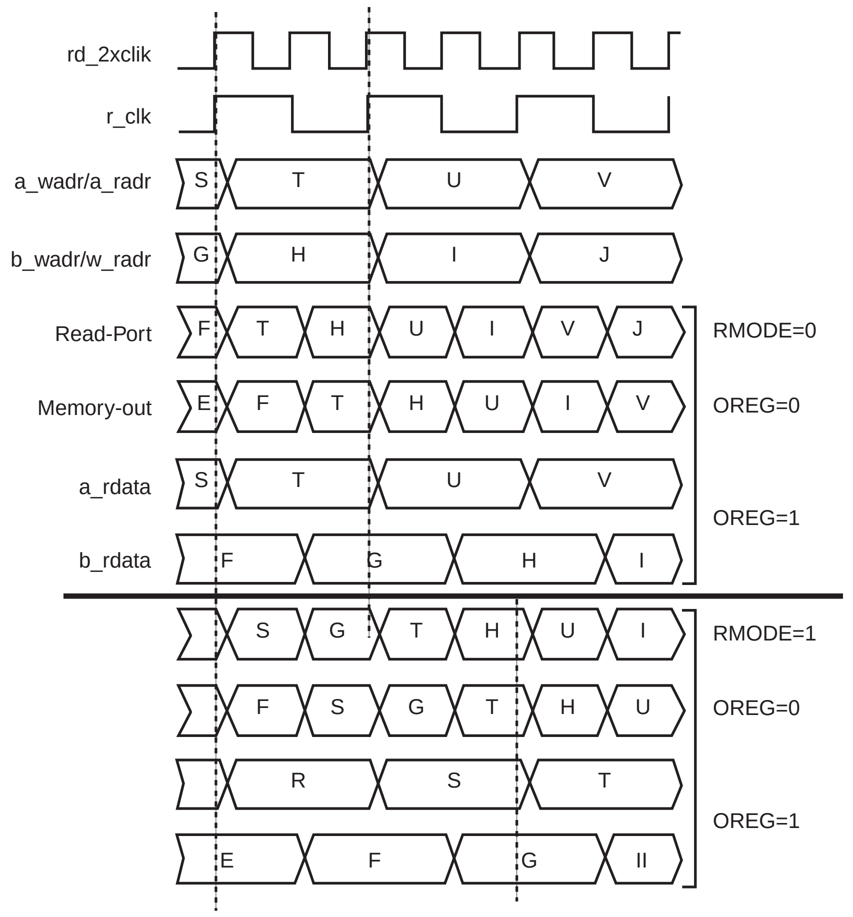

Figure 7 • *Axcelerator Multi-Port Memory Implementation Read Cycle*

Design Considerations

The implementation of both dual-port memory and quad-port memory involves doubling the clock frequency at which data is clocked into the Axcelerator embedded memory and involves using multiplex logic to arbitrate between Port A and Port B. The simplest way to implement the doubled frequency is to make use of the on-chip PLL with the exact configuration generated using the ACTgen Macro Builder. If the configuration has the outputs registered (OREG= ‘1’), this generates flip-flops that will be part of the critical path of the design.

Utilization

Using the reference design example in the “Appendix – Design Example” on page 8, the following tables quantify the additional logic overhead introduced by the necessary gates, flip-flops, and PLL used in both dual-port memory and quad-port memory configurations in unregistered versus registered inputs and outputs configuration with a read depth equal to the write depth.

Dual-Port Memory with Unregistered Inputs and Outputs

Table 3* • *Designer Resource Utilization Report – (PMODE=0, WMODE=0, RMODE=0, OREG=0)

Write Configuration	**4k x 1**	**1k x 4**	**512 x 9**	**128 x 36**
Sequential (R-cells)	0	0	0	0
Combinatorial (C-cells)	15	16	20	45
RAM/FIFO	1	1	1	1
PLL	2	2	2	2

Dual-Port Memory with Registered Inputs and Outputs

Table 4* • *Designer Resource Utilization Report – (PMODE=0, WMODE=1, RMODE=1, OREG=1)

Write Configuration	**4k x 1**	**1k x 4**	**512 x 9**	**128 x 36**
Sequential (R-cells)	34	36	42	164
Combinatorial (C-cells)	17	17	18	53
RAM/FIFO	1	1	1	1
PLL	2	2	2	2

Quad-Port Memory with Unregistered Inputs and Outputs

Table 5* • *Designer Resource Utilization Report – (PMODE=1, WMODE=0, RMODE=0, OREG=0)

Write Configuration	**4k x 1**	**1k x 4**	**512 x 9**	**128 x 36**
Sequential (R-cells)	0	0	0	0
Combinatorial (C-cells)	27	26	29	52
RAM/FIFO	1	1	1	1
PLL	2	2	2	2

Quad-Port Memory with Registered Inputs and Outputs

Table 6* • *Designer Resource Utilization Report – (PMODE=1, WMODE=1, RMODE=1, OREG=1)

Write Configuration	**4k x 1**	**1k x 4**	**512 x 9**	**128 x 36**
Sequential (R-cells)	58	62	78	178
Combinatorial (C-cells)	29	28	31	60
RAM/FIFO	1	1	1	1
PLL	2	2	2	2

Notice the utilization increases significantly from the unregistered inputs and outputs to the registered configuration. This results from the additional flip-flops (sequential R-cells) necessary to generate the registered inputs and outputs for each bit of the data and address busses as well as the enable signals. Also, the utilization shows a slight increase from the dual-port memory to quad-port memory configuration.

Conclusion

Implementation of multi-port memories using a wrapper source code to interface the basic Axcelerator memory block is straightforward and intuitive. Implementation of both dual-port and quad-port memories, although requiring additional logic overhead, which include extra multiplexers and flip-flops, still proves useful in certain designs.

Related Documents

For more information, see the following documents:

Axcelerator Family FPGAs datasheet

http://www.actel.com/documents/AXDS.pdf

Axcelerator Family Memory Blocks

http://www.actel.com/documents/AXblocksAN.pdf

Axcelerator Family PLL and Clock Management

http://www.actel.com/documents/AX_PLL_AN.pdf

A Guide to ACTgen Macros

http://www.actel.com/documents/genguide_UG.pdf

Appendix – Design Example

This design example implements a variable width dual-port or quad-port memory with variable aspect ratio support, based on the memory blocks available in Actel's Axcelerator devices. For deeper memories, the user must cascade blocks together or modify this design example.

Below is a sample instantiation of the multi-port memory wrapper, which may be cut and pasted into the higher-level VHDL code:

```
--  QPM0: mpm_ax
--  GENERIC MAP (PMODE => 0, WMODE => 0, RMODE => 0, OREG => 0,
--               WR_DEPTH => 256, WR_WIDTH => 36, RD_DEPTH => 512);
--
--  PORT MAP    (reset_n  => <your reset>,
--               wr_2xclk => <2x write clock>,
--               wr_clk   => <write port data/enable clock>,
--
--               a_wren    => <active high data enable for write>,
--               a_wadr   => <write/dpm address bus  (8-bits)>,
--               a_wdata  => <write data bus (variable width)>,
--
--               b_wren    => <active high data enable for write>,
--               b_wadr   => <write/dpm address bus  (8-bits)>,
--               b_wdata  => <write data bus (variable width)>,
--
--               rd_2xclk => <2X read clock   (QPM mode)>,
--               rd_clk   => <read data clock (QPM mode)>,
--
--               a_rden    => <output register enable A>,
--               a_radr   => <output address bus 8-bits (QPM mode)>,
--               a_rdata  => <output data bus (variable width)>,
--
--               b_rden    => <output register enable B>,
--               b_radr   => <output address bus 8-bits (QPM mode)>,
--               b_rdata  => <output data bus (variable width)>);
```

Here is the multi-port memory implementation source code:

NOTE: The source code can also be obtained from Actel Technical Support or from your local FAE.

```
-----------------------------------------------------------------------
--
--                    Copyright 2002 Actel Corporation
--
-----------------------------------------------------------------------
--
--                                   mode and bus mis-matches.
-----------------------------------------------------------------------
```

```
--
-- This package declares types for the generic parameters so that
-- you can specify values that convey information on each instance.
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

-- This package contains functions used to translate the
-- generics when calculating bits widths & structure sizes.

package GTYPES is
 function get_adr (DEPTH: INTEGER) return integer;
 function get_max (RDEPTH: INTEGER; WDEPTH: INTEGER) return integer;
 function get_bits(DEPTH: INTEGER) return integer;
 function get_val (DEPTH: INTEGER) return std_logic_vector;
end package GTYPES;

Package body GTYPES is

-- number of valid address bits based on depth (N-1)

 function get_adr(DEPTH: INTEGER) return integer is
  variable result: integer;
 begin
    case (depth) is
     when 128    => result :=  6;
     when 256    => result :=  7;
     when 512    => result :=  8;
     when 1024   => result :=  9;
     when 2048   => result := 10;
     when 4096   => result := 11;
     when others => result := 0;  -- will cause errors
    end case;
  return result;
 end function;

-- Returns the larger of the two values

 function get_max(PMODE: INTEGER; RDEPTH: INTEGER; WDEPTH: INTEGER) return integer is
  variable result: integer;
 begin
   IF (PMODE = 0 AND RDEPTH > WDEPTH) then
      result := get_adr(RDEPTH);
   ELSE
      result := get_adr(WDEPTH);
   END IF;
  return result;
 end function;

-- Width #bits based on selected depth

 function get_bits(DEPTH: INTEGER) return integer is
  variable result: integer;
 begin
    case (depth) is
     when 128    => result := 36;
     when 256    => result := 18;
     when 512    => result := 9;
     when 1024   => result := 4;
     when 2048   => result := 2;
     when 4096   => result := 1;
     when others => result := 0;  -- will cause errors
    end case;
  return result;
 end function;

-- static value to set on RAM for a given depth

 function get_val(DEPTH: INTEGER) return std_logic_vector is
  variable result: std_logic_vector(2 downto 0);
 begin
    case (depth) is
     when 128    => result := "101";
     when 256    => result := "100";
     when 512    => result := "011";
     when 1024   => result := "010";
     when 2048   => result := "001";
     when 4096   => result := "000";
     when others => result := "111";  -- will cause errors
    end case;
  return result;
 end function;
end package body GTYPES;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

use work.GTYPES.all;

-- depending on the tools suite used, the AX library
-- may need to be referenced for VITAL simulation models.
--
-- NOTE: Integer types are used for all GENERIC declarations
-- in order to include synopsys support.

entity mpm_ax is
    generic (PMODE      : INTEGER range 0 to 1 := 0;       -- DPM / QPM
             WMODE      : INTEGER range 0 to 1 := 0;       -- NO  / Pipe-line before mux
             RMODE      : INTEGER range 0 to 1 := 0;       -- NO  / Pipe-line before mux
             OREG       : INTEGER range 0 to 1 := 0;       -- NO  / Pipe-line outputs

             WR_WIDTH   : INTEGER := 36;                   -- 36 ,18 ,9  ,4   ,2   ,1
             WR_DEPTH   : INTEGER range 1 to 4096 256;  -- 128,256,512,1024,2048,4096
             RD_DEPTH   : INTEGER range 1 to 4096 := 512); -- 128,256,512,1024,2048,4096

    port( reset_n   : in  std_logic;  -- active low

          wr_2xclk  : in  std_logic;
          wr_clk    : in  std_logic;

          a_wren    : in  std_logic;
          a_wadr    : in  std_logic_vector(get_max(PMODE, RD_DEPTH, WR_DEPTH) downto 0);
          a_wdata   : in  std_logic_vector(WR_WIDTH - 1 downto 0);

          b_wren    : in  std_logic;
          b_wadr    : in  std_logic_vector(get_max(PMODE, RD_DEPTH, WR_DEPTH) downto 0);
          b_wdata   : in  std_logic_vector(WR_WIDTH - 1 downto 0);

          rd_2xclk  : in  std_logic;
          rd_clk    : in  std_logic;

          a_rden    : in  std_logic;
          a_radr    : in  std_logic_vector(get_adr(RD_DEPTH) downto 0);
          a_rdata   : out std_logic_vector(((WR_WIDTH / get_bits(WR_DEPTH)) *
                                    get_bits(RD_DEPTH)) - 1 downto 0);

          b_rden    : in  std_logic;
          b_radr    : in  std_logic_vector(get_adr(RD_DEPTH) downto 0);
          b_rdata   : out std_logic_vector(((WR_WIDTH / get_bits(WR_DEPTH)) *
                                    get_bits(RD_DEPTH)) - 1 downto 0));
end mpm_ax;

architecture RTL of mpm_ax is

-- AX embedded RAM

component RAM64K36 port(
 WRAD0, WRAD1,  WRAD2,  WRAD3,  WRAD4,  WRAD5,  WRAD6, WRAD7, WRAD8,
 WRAD9, WRAD10, WRAD11, WRAD12, WRAD13, WRAD14, WRAD15 :in std_logic;

 WD0,  WD1,  WD2,  WD3,  WD4,  WD5,  WD6,  WD7,  WD8,  WD9,  WD10,
 WD11, WD12, WD13, WD14, WD15, WD16, WD17, WD18, WD19, WD20, WD21,
 WD22, WD23, WD24, WD25, WD26, WD27, WD28, WD29, WD30, WD31, WD32,
 WD33, WD34, WD35 :in std_logic;

 WW0, WW1, WW2, WEN, WCLK       :in std_logic;
 DEPTH0, DEPTH1, DEPTH2, DEPTH3 :in std_logic;
 RW0, RW1, RW2, REN, RCLK       :in std_logic;

 RDAD0, RDAD1,  RDAD2,  RDAD3,  RDAD4,  RDAD5,  RDAD6, RDAD7, RDAD8,
 RDAD9, RDAD10, RDAD11, RDAD12, RDAD13, RDAD14, RDAD15 :in std_logic;

 RD0,  RD1,  RD2,  RD3,  RD4,  RD5,  RD6,  RD7,  RD8,  RD9,  RD10,
 RD11, RD12, RD13, RD14, RD15, RD16, RD17, RD18, RD19, RD20, RD21,
 RD22, RD23, RD24, RD25, RD26, RD27, RD28, RD29, RD30, RD31, RD32,
 RD33, RD34, RD35 :out std_logic);
end component;

SIGNAL wren_a   : std_logic;
SIGNAL wadr_a   : std_logic_vector(get_max(PMODE, RD_DEPTH, WR_DEPTH) downto 0);
SIGNAL wdata_a  : std_logic_vector(WR_WIDTH - 1 downto 0);
SIGNAL wren_b   : std_logic;
SIGNAL wadr_b   : std_logic_vector(get_max(PMODE, RD_DEPTH, WR_DEPTH) downto 0);
SIGNAL wdata_b  : std_logic_vector(WR_WIDTH - 1 downto 0);
SIGNAL wrenb    : std_logic;
SIGNAL wadr     : std_logic_vector( 11 downto 0);
SIGNAL wdata    : std_logic_vector((36*(WR_WIDTH/get_bits(WR_DEPTH)))-1 downto 0);

SIGNAL rden_a   : std_logic;
SIGNAL rden_b   : std_logic;
SIGNAL rden     : std_logic;
SIGNAL a_oen    : std_logic;
SIGNAL b_oen    : std_logic;

SIGNAL radr_a   : std_logic_vector(get_adr(RD_DEPTH) downto 0);
SIGNAL radr_b   : std_logic_vector(get_adr(RD_DEPTH) downto 0);
SIGNAL radr     : std_logic_vector( 11 downto 0);
SIGNAL rdata    : std_logic_vector((36*(WR_WIDTH/get_bits(WR_DEPTH)))-1 downto 0);
SIGNAL rd_data  : std_logic_vector((((WR_WIDTH / get_bits(WR_DEPTH)) *
                                      get_bits(RD_DEPTH)) - 1) downto 0);

SIGNAL GND         : std_logic;
SIGNAL w_vec,r_vec : std_logic_vector(2 downto 0);
SIGNAL read_clk    : std_logic;

begin

GND <= ’0’; -- used to tie off unused inputs to memories

w_vec <= get_val(WR_DEPTH); -- get control vectors for memories
r_vec <= get_val(RD_DEPTH);

--
-- WRITE PORTS
--

A: if (WMODE = 1) generate
B: process(reset_n, wr_clk)
    begin
     if (reset_n = ’0’) then

      wren_a   <= ’0’;
      wadr_a   <= (OTHERS => ’0’);
      wdata_a  <= (OTHERS => ’0’);

      wren_b   <= ’0’;
      wadr_b   <= (OTHERS => '0');
      wdata_b  <= (OTHERS => '0');

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

      wren_a   <= a_wren  after 1 ns;
      wadr_a   <= a_wadr  after 1 ns;
      wdata_a  <= a_wdata after 1 ns;

      wren_b   <= b_wren  after 1 ns;
      wadr_b   <= b_wadr  after 1 ns;
      wdata_b  <= b_wdata after 1 ns;

     end if;
   end process;
end generate;

--
-- Otherwise, just pass them through to the mux
--

C: if (WMODE /= 1) generate
      wren_a   <= a_wren;
      wadr_a   <= a_wadr;
      wdata_a  <= a_wdata;

      wren_b   <= b_wren;
      wadr_b   <= b_wadr;
      wdata_b  <= b_wdata;
end generate;

--
-- Multiplex the write address/enables to the memory
--

wrenb <= wren_a when (wr_clk = '1') else wren_b;

wadr(get_adr(WR_DEPTH) downto 0) <= wadr_a(get_adr(WR_DEPTH) downto 0)
     when (wr_clk = '1') else wadr_b(get_adr(WR_DEPTH) downto 0);

-- Tie off unused address bits to gnd
```

```
D: if (get_adr(WR_DEPTH) < 11) generate
 wadr(11 downto get_adr(WR_DEPTH)+1) <= (OTHERS => ’0’);
end generate;

-- The write data needs to be "distributed" across
-- the total of 288 possible bits based on WR_DEPTH
-- and allocation of 36 bits of the write bus to
-- each physical memory.  For example, with WR_DEPTH
-- equals 512, only 9 bits of each 36 are connected,
-- for WR_WIDTH of 36, four physical memories are
-- used, so bits 0-8, 36-43, 72-79, and 108-115 are
-- mapped to the 36 wdata_x bits.
--

E: for i in 0 to ((WR_WIDTH/get_bits(WR_DEPTH))-1) generate
   Y: for j in 0 to (get_bits(WR_DEPTH)-1) generate
        wdata((36*i)+j) <= wdata_a((get_bits(WR_DEPTH)*i)+j) when
                        (wr_clk = ’1’) else
                          wdata_b((get_bits(WR_DEPTH)*i)+j);
   end generate;
end generate;

-- Tie off unused data inputs to gnd

F: if (get_bits(WR_DEPTH) < 36) generate
   G: for i in 0 to ((WR_WIDTH/get_bits(WR_DEPTH))-1) generate
      H: for j in (get_bits(WR_DEPTH)) to 35 generate
         wdata((36*i)+j) <= GND;
       end generate;
    end generate;
end generate;

--
-- READ PORTS
--
-- read clock gets wr_2xclk in DPM otherwise, it gets rd_2xclk.
-- Use generates to avoid creating a logic gate here.
--

I: if (PMODE = 0) generate
       read_clk <= wr_2xclk;
end generate;
```

```
J: if (PMODE = 1) generate
       read_clk <= rd_2xclk;
end generate;

--
-- IF QPM and RMODE = 1, register read address inputs.
--

K: if (PMODE = 1 AND RMODE = 1) generate
   L: process(reset_n, rd_clk)
   begin
    if (reset_n = '0') then

      radr_a   <= (OTHERS => '0');
      radr_b   <= (OTHERS => '0');

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

      radr_a   <= a_radr after 1 ns;
      radr_b   <= b_radr after 1 ns;

    end if;
   end process;
end generate;

--
-- Otherwise, if DPM use the write address bits,
-- if QPM, just pass the read address to the mux
--

M: if (PMODE = 0) generate
       radr_a <= a_wadr(get_adr(RD_DEPTH) downto 0);
       radr_b <= b_wadr(get_adr(RD_DEPTH) downto 0);
end generate;

N: if (PMODE = 1 AND RMODE = 0) generate
       radr_a <= a_radr;
       radr_b <= b_radr;
end generate;

-- If QPM and RMODE = 1 or DPM and WMODE = 1
-- register the read enables

Q: if ((PMODE = 1 AND RMODE = 1) OR (PMODE = 0 AND WMODE = 1)) generate
   P: process(reset_n, read_clk)
   begin
    if (reset_n = ’0’) then

      rden_a   <= ’0’;
      rden_b   <= ’0’;

    elsif (read_clk’event and read_clk = ’1’) then

      rden_a   <= a_rden after 1 ns;
      rden_b   <= b_rden after 1 ns;

    end if;
   end process;
end generate;

-- otherwise pass them through

S: if ((PMODE = 0 AND WMODE = 0) OR (PMODE = 1 AND RMODE = 0)) generate
       rden_a <= a_rden;
       rden_b <= b_rden;
end generate;

--
-- Multiplex the read addresses and enables to the memory
-- in DMP mode, use wr_clk, otherwise use rd_clk.
--

S1: if (PMODE = 0) generate
   rden  <= rden_a when (wr_clk = ’1’) else rden_b;
   radr(get_adr(RD_DEPTH) downto 0) <= radr_a(get_adr(RD_DEPTH) downto 0)
         when (wr_clk = ’1’) else radr_b(get_adr(RD_DEPTH) downto 0);
end generate;

S2: if (PMODE = 1) generate
   rden  <= rden_a when (rd_clk = ’1’) else rden_b;
   radr(get_adr(RD_DEPTH) downto 0) <= radr_a(get_adr(RD_DEPTH) downto 0)
         when (rd_clk = ’1’) else radr_b(get_adr(RD_DEPTH) downto 0);
end generate;
```

```
-- Tie off unused inputs

T: if (get_adr(RD_DEPTH) < 11) generate
       radr(11 downto get_adr(RD_DEPTH)+1) <= (OTHERS => ’0’);
end generate;

--
-- The read data needs to be "distributed" across
-- the total of 288 possible bits based on RD_DEPTH
-- and the calculated number of output bits, and
-- allocation of 36 bits of the write bus to each
-- physical memory
--

U: for i in 0 to ((WR_WIDTH/get_bits(WR_DEPTH))-1) generate
    V: for j in 0 to (get_bits(RD_DEPTH)-1) generate
       rd_data((get_bits(RD_DEPTH)*i)+j) <= rdata((i*36)+j);
    end generate;
end generate;

--
-- IF OREG is 1 (Registered), then create the output
-- registers, NOTE: enables are pipe-lined to account
-- for the input registers on the RAM64k36.
-- use rd_clk in QPM mode, wr_clk in DPM mode
--

W: if (PMODE = 1 AND OREG = 1) generate -- QPM

    X: process(reset_n, rd_clk)
    begin
     if (reset_n = ’0’) then
         a_rdata <= (OTHERS => ’0’);
         a_oen   <= ’0’;
     elsif (rd_clk’event and rd_clk = ’1’) then
            a_oen <= rden_a after 1 ns;

      if(a_oen = ’1’) then
         a_rdata <= rd_data(((WR_WIDTH/get_bits(WR_DEPTH))*get_bits(RD_DEPTH)) - 1 downto 0)
after 1 ns;
      end if;
     end if;
    end process;



    Y: process(reset_n, rd_clk)
    begin
     if (reset_n = ’0’) then
         b_rdata <= (OTHERS => ’0’);
         b_oen   <= ’0’;
     elsif (rd_clk’event and rd_clk = ’0’) then
            b_oen <= rden_b after 1 ns;

      if(b_oen = ’1’) then
         b_rdata <= rd_data(((WR_WIDTH/get_bits(WR_DEPTH))*get_bits(RD_DEPTH)) - 1 downto 0)
after 1 ns;
      end if;
     end if;
    end process;

end generate;

--
--  In dual port mode - use the write clock for output registers.
--

Z: if (PMODE = 0 AND OREG = 1) generate

    AA: process(reset_n, wr_clk)
    begin
     if (reset_n = ’0’) then
         a_rdata <= (OTHERS => ’0’);
         a_oen   <= ’0’;
     elsif (wr_clk’event and wr_clk = ’1’) then
            a_oen <= rden_a after 1 ns;

      if(a_oen = ’1’) then
         a_rdata <= rd_data(((WR_WIDTH/get_bits(WR_DEPTH))*get_bits(RD_DEPTH)) - 1 downto 0)
after 1 ns;
      end if;
     end if;
    end process;
    AB: process(reset_n, wr_clk)
    begin
     if (reset_n = ’0’) then
         b_rdata <= (OTHERS => ’0’);
         b_oen   <= ’0’;
     elsif (wr_clk’event and wr_clk = ’0’) then
            b_oen <= rden_b after 1 ns;

      if(b_oen = ’1’) then
         b_rdata <= rd_data(((WR_WIDTH/get_bits(WR_DEPTH))*get_bits(RD_DEPTH)) - 1 downto 0)
after 1 ns;
      end if;
     end if;
    end process;

end generate;

--
-- otherwise, assign the ouput of the
-- memory directly to the read data ports.
--

AC: if (OREG /= 1) generate
    a_rdata <= rd_data(((WR_WIDTH/get_bits(WR_DEPTH))*get_bits(RD_DEPTH)) - 1 downto 0); --
assign to output ports
    b_rdata <= rd_data(((WR_WIDTH/get_bits(WR_DEPTH))*get_bits(RD_DEPTH)) - 1 downto 0);
end generate;

--
-- Generate RAM BLOCK(s)
--

AD: for i in 0 to ((WR_WIDTH/get_bits(WR_DEPTH))-1) generate
      W:RAM64K36 port map(
        WRAD0 => wadr(0), WRAD1  => wadr(1),  WRAD2  => wadr(2),
        WRAD3 => wadr(3), WRAD4  => wadr(4),  WRAD5  => wadr(5),
        WRAD6 => wadr(6), WRAD7  => wadr(7),  WRAD8  => wadr(8),
        WRAD9 => wadr(9), WRAD10 => wadr(10), WRAD11 => wadr(11),
        WRAD12=> GND, WRAD13 => GND, WRAD14 => GND, WRAD15 => GND,

        WD0 => wdata((i*36)+ 0),  WD1 => wdata((i*36)+ 1),
        WD2 => wdata((i*36)+ 2),  WD3 => wdata((i*36)+ 3),
        WD4 => wdata((i*36)+ 4),  WD5 => wdata((i*36)+ 5),
        WD6 => wdata((i*36)+ 6),  WD7 => wdata((i*36)+ 7),
        WD8 => wdata((i*36)+ 8),  WD9 => wdata((i*36)+ 9),
        WD10=> wdata((i*36)+10),  WD11=> wdata((i*36)+11),
        WD12=> wdata((i*36)+12),  WD13=> wdata((i*36)+13),
        WD14=> wdata((i*36)+14),  WD15=> wdata((i*36)+15),
        WD16=> wdata((i*36)+16),  WD17=> wdata((i*36)+17),
        WD18=> wdata((i*36)+18),  WD19=> wdata((i*36)+19),
        WD20=> wdata((i*36)+20),  WD21=> wdata((i*36)+21),
        WD22=> wdata((i*36)+22),  WD23=> wdata((i*36)+23),
        WD24=> wdata((i*36)+24),  WD25=> wdata((i*36)+25),
        WD26=> wdata((i*36)+26),  WD27=> wdata((i*36)+27),
        WD28=> wdata((i*36)+28),  WD29=> wdata((i*36)+29),
        WD30=> wdata((i*36)+30),  WD31=> wdata((i*36)+31),
        WD32=> wdata((i*36)+32),  WD33=> wdata((i*36)+33),
        WD34=> wdata((i*36)+34),  WD35=> wdata((i*36)+35),

        WW0 => w_vec(0), WW1 => w_vec(1), WW2 => w_vec(2),
        WEN => wrenb, WCLK => wr_2xclk,

        DEPTH0 => GND, DEPTH1 => GND, DEPTH2 => GND, DEPTH3 => GND,

        RW0 => r_vec(0), RW1 => r_vec(1), RW2 => r_vec(2),
        REN => rden, RCLK => read_clk,

        RDAD0 => radr(0), RDAD1  => radr(1),  RDAD2  => radr(2),
        RDAD3 => radr(3), RDAD4  => radr(4),  RDAD5  => radr(5),
        RDAD6 => radr(6), RDAD7  => radr(7),  RDAD8  => radr(8),
        RDAD9 => radr(9), RDAD10 => radr(10), RDAD11 => radr(11),
        RDAD12=> GND, RDAD13 => GND, RDAD14 => GND, RDAD15 => GND,

        RD0 => rdata((i*36)+ 0),  RD1 => rdata((i*36)+ 1),
        RD2 => rdata((i*36)+ 2),  RD3 => rdata((i*36)+ 3),
        RD4 => rdata((i*36)+ 4),  RD5 => rdata((i*36)+ 5),
        RD6 => rdata((i*36)+ 6),  RD7 => rdata((i*36)+ 7),
        RD8 => rdata((i*36)+ 8),  RD9 => rdata((i*36)+ 9),
        RD10=> rdata((i*36)+10),  RD11=> rdata((i*36)+11),
        RD12=> rdata((i*36)+12),  RD13=> rdata((i*36)+13),
        RD14=> rdata((i*36)+14),  RD15=> rdata((i*36)+15),
        RD16=> rdata((i*36)+16),  RD17=> rdata((i*36)+17),
        RD18=> rdata((i*36)+18),  RD19=> rdata((i*36)+19),
        RD20=> rdata((i*36)+20),  RD21=> rdata((i*36)+21),
        RD22=> rdata((i*36)+22),  RD23=> rdata((i*36)+23),
        RD24=> rdata((i*36)+24),  RD25=> rdata((i*36)+25),
        RD26=> rdata((i*36)+26),  RD27=> rdata((i*36)+27),
        RD28=> rdata((i*36)+28),  RD29=> rdata((i*36)+29),
        RD30=> rdata((i*36)+30),  RD31=> rdata((i*36)+31),
        RD32=> rdata((i*36)+32),  RD33=> rdata((i*36)+33),
        RD34=> rdata((i*36)+34),  RD35=> rdata((i*36)+35));
end generate;

end RTL;
```

http://www.actel.com

Actel Corporation
955 East Arques Avenue
Sunnyvale, California 94086
USA
Tel: (408) 739-1010
Fax: (408) 739-1540

Actel Europe Ltd.
Dunlop House, Riverside Way
Camberley, Surrey GU15 3YL
United Kingdom
Tel: +44 (0)1276 401450
Fax: +44 (0)1276 401490

Actel Japan
EXOS Ebisu Bldg. 4F
1-24-14 Ebisu Shibuya-ku
Tokyo 150 Japan
Tel: +81 03-3445-7671
Fax: +81 03-3445-7668

Actel Hong Kong
39th Floor
One Pacific Place
88 Queensway
Admiralty, Hong Kong
Tel: 852-22735712

51900038-0/7.03