Implementing Multi-Port Memories in ProASIC^PLUS Devices

Table of Contents

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 Microsemi ProASIC^PLUS® field programmable gate array (FPGA) devices.

The ProASIC^PLUS device architecture contains embedded SRAM cells that can be configured as static memory blocks with independent read and write ports. Each basic memory block has a size of 256 words by 9 bits with a single data port interface. For additional details on embedded memory blocks in ProASIC^PLUS devices, refer to *ProASICPLUS Flash Family FPGAs Datasheet* or *AC281: ProASICPLUS RAM-FIFO Blocks Application Note*.

Figure 1 shows a block diagram of the basic memory block.

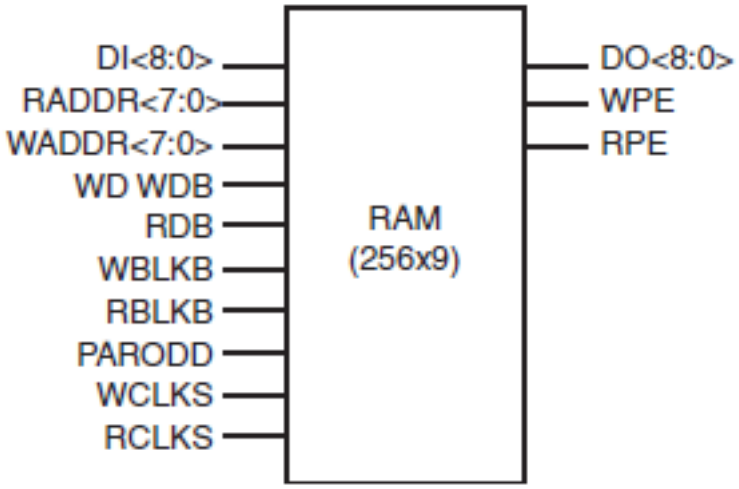

***Figure 1* • Basic Embedded SRAM Memory Block Structure**

The embedded memory blocks in ProASIC^PLUS 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) are available for simultaneous read and write operations into the ProASIC^PLUS embedded SRAM blocks. 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 ProASIC^PLUS devices involve the use of an additional clock operating at double the read and write frequency to access the memory space through some multiplex logic and arbitrate between the data access ports. The overall bandwidth of the memory (bit or bits) remains the same, and the only difference between the single and the multi-port memory is the read/write frequency versus data width-trade off.

Although more than one data access port is now available, they share the same memory space. Simultaneous read/write cycles to the same memory address result in reading the pre-existing memory contents followed by the memory being updated with the new data at the end of the clock cycle.

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 3 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 2 shows the corresponding ports of a dual-port memory block.

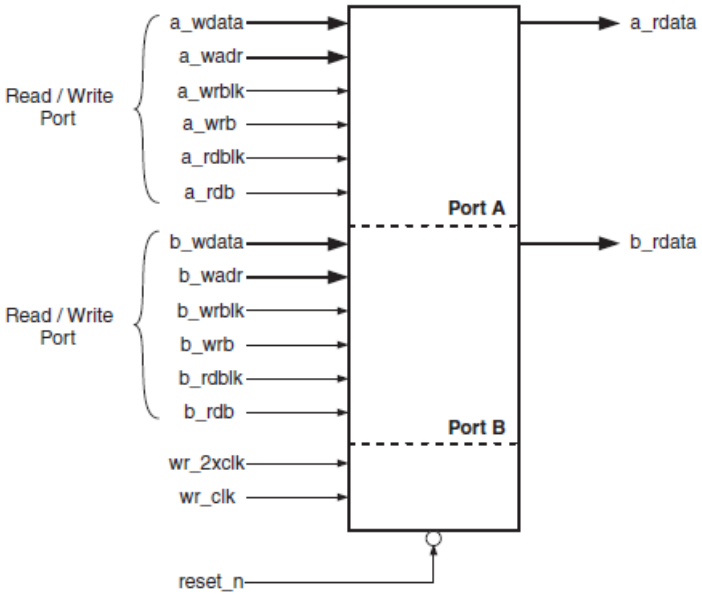

***Figure 2* • Dual-Port Memory Block Interface Signals**

Figure 3 shows the corresponding ports of a quad-port memory block.

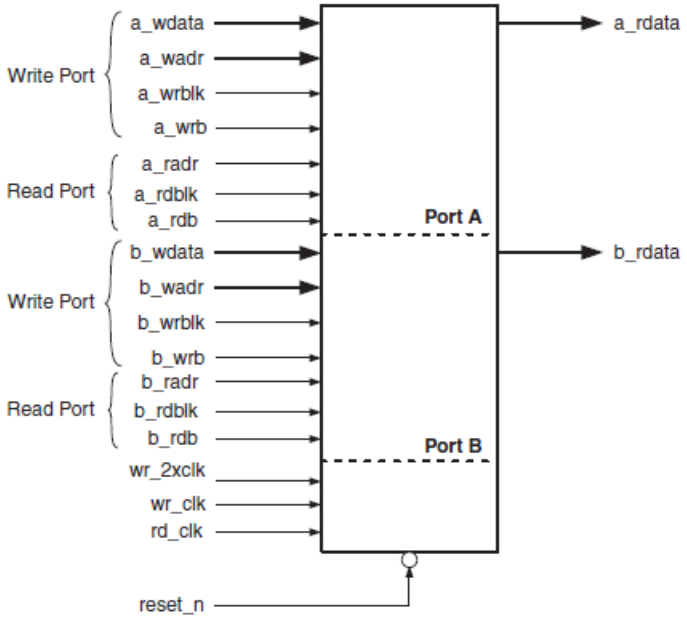

***Figure 3* • Quad-Port Memory Block Interface Signals**

Table 1 summarizes the interface signals of the memory block.

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

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

Implementing Multi-Port Memories

In the referenced example ("Appendix: Design Example" on page 10), 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 (that is 256 words), but the width is variable up to 72 bits. The ProASIC^{PLUS} RAM256X9SA macro is used as the basic memory block for this wrapper.

Since implementation of multi-port memories relies on the embedded memory block being clocked at twice the data clock rate, a double frequency clock needs to be generated. The original data clock input (wr_clk) is easily doubled in frequency to generate the required wr_2xclk signal using the PLLs in ProASIC^{PLUS} architecture. For more information on how to generate a PLL for ProASIC^{PLUS} devices, refer to *Microsemi SmartGen, FlashROM, ASB, and Flash Memory System Builder User Guide* or *AC306: Using ProASICPLUS Clock Conditioning Circuits Application Note*.

Table 2 describes the configurable parameters for the reference design in the "Appendix: Design Example" on page 10.

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

Parameter	Value	Description
PMODE	0 (default)	Dual-port memory configuration
	1	Quad-port memory configuration
PIPE	0 (default)	Inputs not registered, just multiplex logic for inputs
	1	Register inputs, then multiplex inputs to memory
OREG	0 (default)	Transparent output mode
	1	Registered output mode
WIDTH	1:72 (default = 9)	Number of data + parity bits

Read Ports: Dual-Port Memory

A block diagram of the dual-port memory implementation is shown in Figure 4.

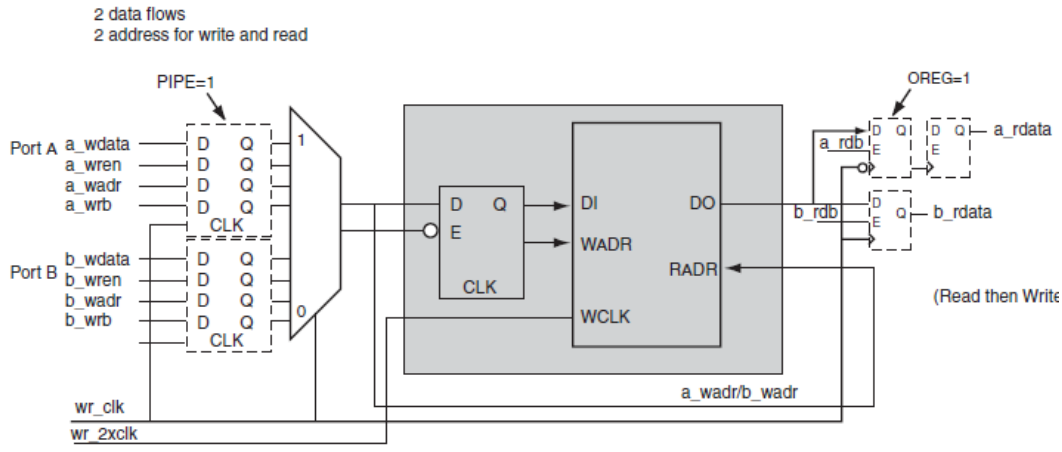

Figure 4 • **Dual-Port Memory Implementation**

In this configuration, the write addresses and write clock are used to read from the memory. The read address inputs and read clock remain unused in the code. If OREG = 0, the data outputs propagate directly to both of the data output ports; otherwise, when OREG = 1, the output is pipelined with the rising-edge of wr_clk. Data for the Read/Write Port A is registered on the falling-edge of wr_clk and then re-timed to the next rising-edge, while data for Read/Write Port B is registered on the rising-edge of wr_clk. This resynchronizes the data on Port A and Port B so that the apparent operation of the memory is to read both ports simultaneously. The memory read operation supported by this wrapper is asynchronous. The read enable inputs are used as inputs to enable the output registers.

Read Ports: Quad-Port Memory

A block diagram of the quad-port memory implementation is shown in Figure 5.

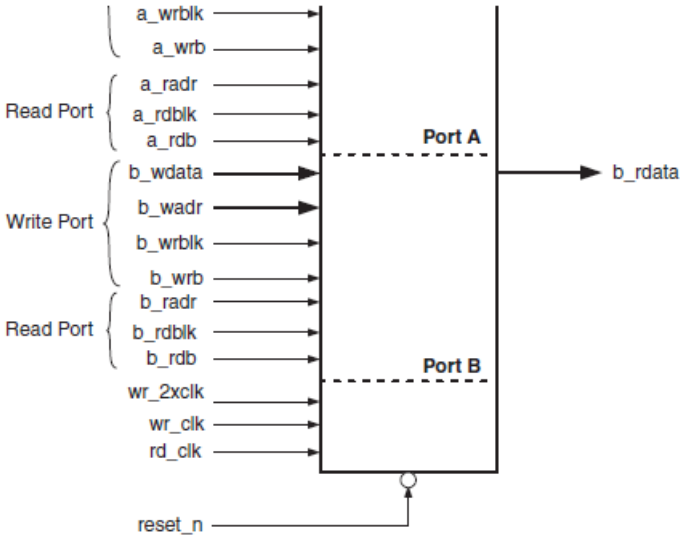

***Figure 5* • Quad-Port Memory Implementation**

If PIPE is set to 1, the address and enable inputs for both read ports are registered with the rising edge of the read clock rd_clk. Then, the read address and read enable inputs for Port A and Port B are multiplexed to the memory and settle, while the rd_clk signal is high and low, respectively. If PIPE 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 ProASIC^PLUS memory.

If OREG = 0, the data outputs propagate directly to the data output ports; otherwise when OREG = 1, the output is resynchronized with the rising-edge of rd_clk. Read data for Read Port A is registered on the falling-edge of rd_clk and then re-timed to the next rising-edge, and read data for Read Port B is registered on the rising-edge of rd_clk. This resynchronizes the data on Port A and Port B so that the apparent operation of the memory is to read both ports simultaneously. The memory read operation supported by this wrapper is asynchronous. The read enable inputs are used as inputs 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 PIPE = 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).

Timing Diagrams

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

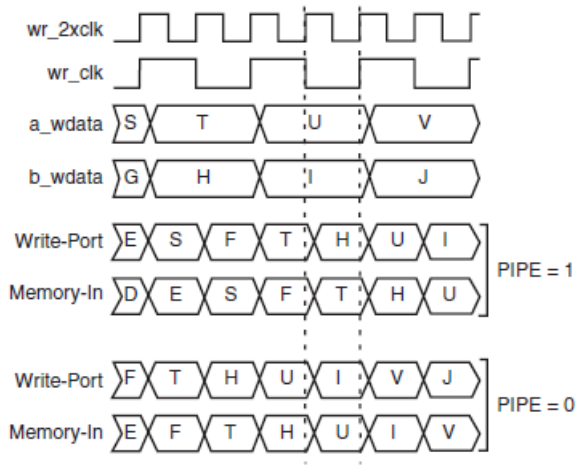

***Figure 6* • ProASIC^PLUS Multi-Port memory Implementation Write Cycle**

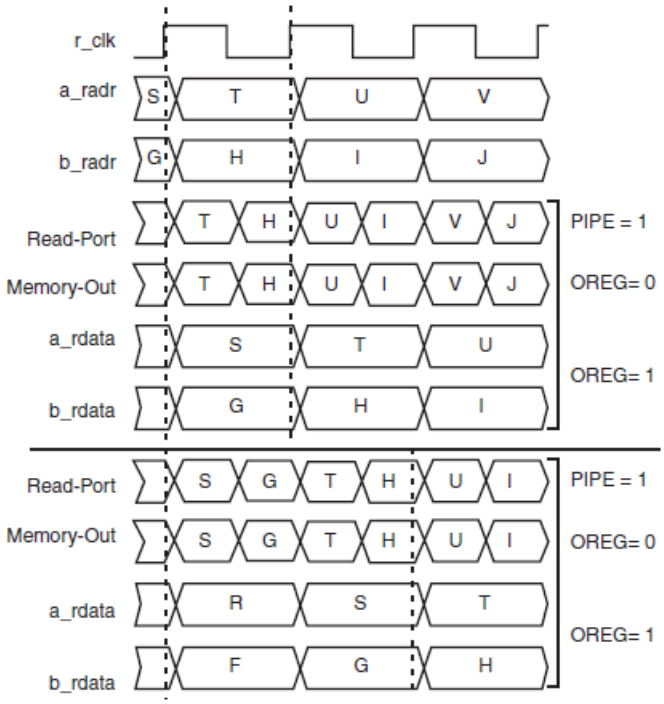

***Figure 7* • ProASIC^PLUS Multi-Port memory Implementation Read Cycle**

Design Considerations

The implementation of both dual-port memory and quad-port memory involves doubling the clock frequency at which data is clocked into ProASIC^PLUS embedded memory and the use of 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 ACTgen Macro Builder.

Utilization

Using the reference design example in the "Appendix: Design Example" on page 10, the following tables quantify the additional logic overhead introduced by the necessary gates, flip-flops, and PLL used in both dual-port and quad-port memory configurations in un-registered versus registered inputs and outputs configuration.

The limiting portion of the design is the use of a doubled-frequency clock to a read/write into memory. The maximum PLL output frequency is 180 MHz, as listed on *ProASICPLUS Flash Family FPGAs Datasheet*. Therefore; if PLL is used to generate the doubled-frequency clock, the operation of the wrapper code is at a limit of 90 MHz read/write. To achieve faster performance, the double-frequency clock can be generated off chip.

If the configuration has the outputs registered (OREG = 1), this generates opposite edge flip-flops that is part of the critical path of the design.

Dual-Port Memory with Unregistered Inputs and Outputs.

Table 3 • **Designer Resource Utilization Report (PMODE = 0, PIPE = 0, OREG = 0)**

Data Width	9 Bits	18 Bits	36 Bits	72 Bits
Core Cells	19	28	46	82
RAM/FIFO Cells	1	2	4	8
PLLs	1	1	1	1

Dual-Port Memory with Registered Inputs and Outputs.

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

Data Width	9 Bits	18 Bits	36 Bits	72 Bits
Core Cells	114	195	356	680
RAM/FIFO Cells	1	2	4	8
PLLs	1	1	1	1

Quad-Port Memory with Unregistered Inputs and Outputs.

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

Data Width	9 Bits	18 Bits	36 Bits	72 Bits
Core Cells	27	36	54	90
RAM/FIFO Cells	1	2	4	8
PLLs	1	1	1	1

Quad-Port Memory with Registered Inputs and Outputs.

Table 6 • **Designer Resource Utilization Report (PMODE = 1, PIPE = 1, OREG = 1)**

Data Width	9 Bits	18 Bits	36 Bits	72 Bits
Core Cells	142	223	384	708
RAM/FIFO Cells	1	2	4	8
PLLs	1	1	1	1

Notice the utilization increases significantly from the unregistered inputs and outputs to the registered configuration. This is due to the additional flip-flops 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 to quad-port memory configuration.

Conclusion

Implementation of multi-port memories using a wrapper source code to interface the basic ProASIC^{PLUS} memory block is straightforward and intuitive. Although, implementation of both dual-port and quad-port memories requires additional logic overhead, including extra multiplexers, and flip-flops, still proves useful in certain designs.

While the particular reference design included in the "Appendix: Design Example" on page 10 does not account for parity, parity input and output signals can be easily implemented in the multi-port memory wrapper source code. First, make use of the parity checking/generating capabilities built-in to the ProASIC^{PLUS} memory blocks, and instantiate the desired memory macro in place of the RAM256x9SA used in the example. Then, follow the basic principle of multiplexing Port A and Port B parity signals at rising and falling edges of the clock signal.

Related Documents

For more information, see the following documents:

- ProASICPLUS Flash Family FPGAs Datasheet
- *AC281: ProASICPLUS RAM-FIFO Blocks Application Note*
- *AC306: Using ProASICPLUS Clock Conditioning Circuits Application Note*
- *Microsemi SmartGen, FlashROM, ASB, and Flash Memory System Builder User Guide*

Appendix: Design Example

This design example implements a variable width dual-port or quad-port memory (up to 72 bits wide), based on the 256 × 9 memory blocks available in Microsemi ProASIC^PLUS devices. For deeper memories, the user must gang these blocks together and modify this design example.

A sample instantiation of the multi-port memory wrapper, which may be cut and pasted into the higher-level VHDL code are as follows:

```
-- QPM0: mpm_apa
-- GENERIC MAP (PMODE => 0, PIPE = 0, OREG = 0, WIDTH = 9);
--
-- PORT MAP (reset_n => <your reset>,
-- wr_2xclk => <2x write clock>,
-- wr_clk => <write port data/enable clock>,
--
-- a_wrblk => <active low block enable for write>,
-- a_wrb => <active low data enable for write>,
-- a_wadr => <write/dpm address bus (8-bits)>,
-- a_wdata => <write data bus (variable width)>,
--
-- b_wrblk => <active low block enable for write>,
-- b_wrb => <active low data enable for write>,
-- b_wadr => <write/dpm address bus (8-bits)>,
-- b_wdata => <write data bus (variable width)>,
--
-- rd_clk => <read data clock (QPM mode)>,
--
-- a_rdblk => <output register enable 1>,
-- a_rdb => <output register enable 2>,
-- a_radr => <output address bus 8-bits (QPM mode)>,
-- a_rdata => <output data bus (variable width)>,
--
-- b_rdblk => <output register enable 1>,
-- b_rdb => <output register enable 2>,
-- b_radr => <output address bus 8-bits (QPM mode)>,
-- b_rdata => <output data bus (variable width)>);
```

The multi-port memory implementation source code are as follows:

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

```
------------------------------------------------------------------------
--
-- Copyright 2002 Microsemi Corporation
--
------------------------------------------------------------------------
--

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

-- depending on the tools suite used, the APA library
-- may need to be referenced for VITAL simulation models.

--library apa;
--use apa.all;
```

```
-- NOTE: Integer types are used for all GENERIC declarations
-- in order to include synopsys support.

entity mpm_apa is
generic (PMODE : INTEGER range 0 to 1 := 0; -- DPM, QPM
PIPE : INTEGER range 0 to 1 := 0; -- M, R
OREG : INTEGER range 0 to 1 := 0; -- M, R
WIDTH : INTEGER range 1 to 72 := 9); -- 1:72

port( reset_n : in std_logic; -- active low

wr_2xclk : in std_logic;
wr_clk : in std_logic;

a_wrblkb : in std_logic;
a_wrb : in std_logic;
a_wadr : in std_logic_vector(7 downto 0);
a_wdata : in std_logic_vector(WIDTH - 1 downto 0);

b_wrblkb : in std_logic;
b_wrb : in std_logic;
b_wadr : in std_logic_vector(7 downto 0);
b_wdata : in std_logic_vector(WIDTH - 1 downto 0);

rd_clk : in std_logic;

a_rdblkb : in std_logic;
a_rdb : in std_logic;
a_radr : in std_logic_vector(7 downto 0);
a_rdata : out std_logic_vector(WIDTH - 1 downto 0);

b_rdblkb : in std_logic;
b_rdb : in std_logic;
b_radr : in std_logic_vector(7 downto 0);
b_rdata : out std_logic_vector(WIDTH - 1 downto 0));

end mpm_apa;

architecture RTL of mpm_apa is

-- APA 256x9 RAM with parity checking (parity not used)

component RAM256x9SA
port(
DO8 :out STD_ULOGIC;
DO7 :out STD_ULOGIC;
DO6 :out STD_ULOGIC;
DO5 :out STD_ULOGIC;
DO4 :out STD_ULOGIC;
DO3 :out STD_ULOGIC;
DO2 :out STD_ULOGIC;
DO1 :out STD_ULOGIC;
DO0 :out STD_ULOGIC;
```

```
WPE :out STD_ULOGIC;
RPE :out STD_ULOGIC;
DOS :out STD_ULOGIC;
WADDR7 :in STD_ULOGIC;
WADDR6 :in STD_ULOGIC;
WADDR5 :in STD_ULOGIC;
WADDR4 :in STD_ULOGIC;
WADDR3 :in STD_ULOGIC;
WADDR2 :in STD_ULOGIC;
WADDR1 :in STD_ULOGIC;
WADDR0 :in STD_ULOGIC;
RADDR7 :in STD_ULOGIC;
RADDR6 :in STD_ULOGIC;
RADDR5 :in STD_ULOGIC;
RADDR4 :in STD_ULOGIC;
RADDR3 :in STD_ULOGIC;
RADDR2 :in STD_ULOGIC;
RADDR1 :in STD_ULOGIC;
RADDR0 :in STD_ULOGIC;
WCLKS :in STD_ULOGIC;
DI8 :in STD_ULOGIC;
DI7 :in STD_ULOGIC;
DI6 :in STD_ULOGIC;
DI5 :in STD_ULOGIC;
DI4 :in STD_ULOGIC;
DI3 :in STD_ULOGIC;
DI2 :in STD_ULOGIC;
DI1 :in STD_ULOGIC;
DI0 :in STD_ULOGIC;
WRB :in STD_ULOGIC;
RDB :in STD_ULOGIC;
WBLKB :in STD_ULOGIC;
RBLKB :in STD_ULOGIC;
PARODD :in STD_ULOGIC;
DIS :in STD_ULOGIC);
end component;

SIGNAL wrblk_a : std_logic;
SIGNAL wrb_a : std_logic;
SIGNAL wadr_a : std_logic_vector(7 downto 0);
SIGNAL wdata_a : std_logic_vector(WIDTH - 1 downto 0);
SIGNAL wrblk_b : std_logic;
SIGNAL wrb_b : std_logic;
SIGNAL wadr_b : std_logic_vector(7 downto 0);
SIGNAL wdata_b : std_logic_vector(WIDTH - 1 downto 0);
SIGNAL wrblk : std_logic;
SIGNAL wrb : std_logic;
SIGNAL wadr : std_logic_vector(7 downto 0);
SIGNAL wdata : std_logic_vector(80 downto 0);

SIGNAL rdblkb_a : std_logic;
SIGNAL rdblkb_b : std_logic;
SIGNAL rdb_a : std_logic;
SIGNAL rdb_b : std_logic;
```

```
SIGNAL rdblkb : std_logic;
SIGNAL rdb : std_logic;

SIGNAL radr_a : std_logic_vector(7 downto 0);
SIGNAL radr_b : std_logic_vector(7 downto 0);
SIGNAL radr : std_logic_vector(7 downto 0);
SIGNAL rd_data : std_logic_vector(80 downto 0);

SIGNAL a_idata : std_logic_vector(WIDTH - 1 downto 0);

SIGNAL GND : std_logic;

begin

begin
GND <= '0'; -- used to tie off unused inputs to memories
--
-- Register incoming data from the write ports
--
A: if (PIPE = 1) generate
B: process(reset_n, wr_clk)
begin
if (reset_n = '0') then

wrblk_a <= '0';
wrb_a <= '0';
wadr_a <= (OTHERS => '0');
wdata_a <= (OTHERS => '0');

wrblk_b <= '0';
wrb_b <= '0';
wadr_b <= (OTHERS => '0');
wdata_b <= (OTHERS => '0');

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

wrblk_a <= a_wrblkb after 1 ns;
wrb_a <= a_wrb after 1 ns;
wadr_a <= a_wadr after 1 ns;
wdata_a <= a_wdata after 1 ns;

wrblk_b <= b_wrblkb after 1 ns;
wrb_b <= b_wrb 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 (PIPE /= 1) generate
```

```
wrblk_a <= a_wrblkb;
wrb_a <= a_wrb;
wadr_a <= a_wadr;
wdata_a <= a_wdata;

wrblk_b <= b_wrblkb;
wrb_b <= b_wrb;
wadr_b <= b_wadr;
wdata_b <= b_wdata;
end generate;

--
-- Multiplex the write ports to the memory
--

wrblk <= wrblk_a when (wr_clk = '1') else wrblk_b;
wrb <= wrb_a when (wr_clk = '1') else wrb_b;
wadr <= wadr_a when (wr_clk = '1') else wadr_b;

wdata(80 downto WIDTH) <= (OTHERS => '0'); -- tie off unused bits
wdata(WIDTH - 1 downto 0) <= wdata_a when (wr_clk = '1') else
wdata_b(WIDTH - 1 downto 0);

--
-- IF FOUR-Port and PIPE = 1 Register the read
-- addresses and enables
--

D: if (PMODE = 1 AND PIPE = 1) generate
E: process(reset_n, rd_clk)
begin
if (reset_n = '0') then

rdblkb_a <= '0';
rdb_a <= '0';
radr_a <= (OTHERS => '0');

rdblkb_b <= '0';
rdb_b <= '0';
radr_b <= (OTHERS => '0');

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

rdblkb_a <= a_rdblkb after 1 ns;
rdb_a <= a_rdb after 1 ns;
radr_a <= a_radr after 1 ns;

rdblkb_b <= b_rdblkb after 1 ns;
rdb_b <= b_rdb after 1 ns;
radr_b <= b_radr after 1 ns;

end if;
end process;
end generate;
```

```
--
-- Otherwise, it's just a pass them through
--

F: if (PIPE /= 1) generate
rdblkb_a <= a_rdblkb;
rdb_a <= a_rdb;
radr_a <= a_radr;

rdblkb_b <= b_rdblkb;
rdb_b <= b_rdb;
radr_b <= b_radr;
end generate;

--
-- In four-port mode, multiplex the read addresses
-- NOTE: enables are used for output registers only.
--
W: if (PMODE = 1) generate
radr <= radr_a when (rd_clk = '1') else radr_b;
end generate;

--
-- IF OREG is 1 (registered), then create the output
-- registers, use rd_clk in QPM (PMODE = 1) mode...
--

rd_data(80 downto WIDTH) <= (OTHERS => '0'); -- tie off unused bits.

G: if (PMODE = 1 AND OREG = 1) generate

I: process(reset_n, rd_clk)
begin
if (reset_n = '0') then
a_idata <= (OTHERS => '0');
elsif (rd_clk'event and rd_clk = '0') then
if(rdblkb_a = '0' AND rdb_a = '0') then
a_idata <= rd_data(WIDTH-1 downto 0) after 1 ns;
end if;
end if;
end process;

J: process(reset_n, rd_clk)
begin
if (reset_n = '0') then
b_rdata <= (OTHERS => '0');
a_rdata <= (OTHERS => '0');
elsif (rd_clk'event and rd_clk = '1') then
if(rdblkb_b = '0' AND rdb_b = '0') then
b_rdata <= rd_data(WIDTH-1 downto 0) after 1 ns;
end if;
if(rdblkb_a = '0' AND rdb_a = '0') then
a_rdata <= a_idata after 1 ns;
```

```
end if;
end if;
end process;

end generate;

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

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

M: process(reset_n, wr_clk)
begin
if (reset_n = '0') then
a_idata <= (OTHERS => '0');
elsif (wr_clk'event and wr_clk = '0') then
if(a_rdblkb = '0' AND a_rdb = '0') then
a_idata <= rd_data(WIDTH-1 downto 0) after 1 ns;
end if;
end if;
end process;

N: process(reset_n, wr_clk)
begin
if (reset_n = '0') then
b_rdata <= (OTHERS => '0');
a_rdata <= (OTHERS => '0');
elsif (wr_clk'event and wr_clk = '1') then
if(b_rdblkb = '0' AND b_rdb = '0') then
b_rdata <= rd_data(WIDTH-1 downto 0) after 1 ns;
end if;
if(a_rdblkb = '0' AND a_rdb = '0') then
a_rdata <= a_idata after 1 ns;
end if;
end if;
end process;

end generate;

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

O: if (OREG /= 1) generate
a_rdata <= rd_data(WIDTH-1 downto 0); -- assign to output ports
b_rdata <= rd_data(WIDTH-1 downto 0);
end generate;

--
-- IF QPM generate (WIDTH/9) RAM BLOCKS with separate read/write ports
--
Q: if (PMODE = 1) generate
```

```
R: for i in 0 to (WIDTH/9) generate
S: RAM256x9SA
port map( DI8 => wdata((i*9)+8), DI7 => wdata((i*9)+7),
DI6 => wdata((i*9)+6), DI5 => wdata((i*9)+5),
DI4 => wdata((i*9)+4), DI3 => wdata((i*9)+3),
DI2 => wdata((i*9)+2), DI1 => wdata((i*9)+1),
DI0 => wdata((i*9)+0),

WADDR7 => wadr(7), WADDR6 => wadr(6),
WADDR5 => wadr(5), WADDR4 => wadr(4),
WADDR3 => wadr(3), WADDR2 => wadr(2),
WADDR1 => wadr(1), WADDR0 => wadr(0),

WBLKB => wrblk, WRB => wrb,
WCLKS => wr_2xclk,

WPE => open, RPE => open, DOS => open,

RBLKB => GND, RDB => GND,

RADDR7 => radr(7), RADDR6 => radr(6),
RADDR5 => radr(5), RADDR4 => radr(4),
RADDR3 => radr(3), RADDR2 => radr(2),
RADDR1 => radr(1), RADDR0 => radr(0),

DO8 => rd_data((i*9)+8), DO7 => rd_data((i*9)+7),
DO6 => rd_data((i*9)+6), DO5 => rd_data((i*9)+5),
DO4 => rd_data((i*9)+4), DO3 => rd_data((i*9)+3),
DO2 => rd_data((i*9)+2), DO1 => rd_data((i*9)+1),
DO0 => rd_data((i*9)+0),
PARODD => GND, DIS => GND

);
end generate;
end generate;

--
-- IF DPM generate (WIDTH/9) RAM BLOCKS with combined read/write ports
-- ie: use muxed write address for the read address also.
--

T: if (PMODE = 0) generate
U: for i in 0 to (WIDTH/9) generate
V: RAM256x9SA
port map( DI8 => wdata((i*9)+8), DI7 => wdata((i*9)+7),
DI6 => wdata((i*9)+6), DI5 => wdata((i*9)+5),
DI4 => wdata((i*9)+4), DI3 => wdata((i*9)+3),
DI2 => wdata((i*9)+2), DI1 => wdata((i*9)+1),
DI0 => wdata((i*9)+0),

WADDR7 => wadr(7), WADDR6 => wadr(6),
WADDR5 => wadr(5), WADDR4 => wadr(4),
WADDR3 => wadr(3), WADDR2 => wadr(2),
WADDR1 => wadr(1), WADDR0 => wadr(0),
```

```
WBLKB => wrblk, WRB => wrb,
WCLKS => wr_2xclk,

WPE => open, RPE => open, DOS => open,

RBLKB => GND, RDB => GND,

RADDR7 => wadr(7), RADDR6 => wadr(6),
RADDR5 => wadr(5), RADDR4 => wadr(4),
RADDR3 => wadr(3), RADDR2 => wadr(2),
RADDR1 => wadr(1), RADDR0 => wadr(0),

DO8 => rd_data((i*9)+8), DO7 => rd_data((i*9)+7),
DO6 => rd_data((i*9)+6), DO5 => rd_data((i*9)+5),
DO4 => rd_data((i*9)+4), DO3 => rd_data((i*9)+3),
DO2 => rd_data((i*9)+2), DO1 => rd_data((i*9)+1),
DO0 => rd_data((i*9)+0),

PARODD => GND, DIS => GND
);
end generate;
end generate;
end RTL;
```

List of Changes

The following table shows important changes made in this document for each revision.

Revision	Changes	Page
Revision 1 (June 2016)	Non-technical updates.	N/A
Revision 0 (July 2003)	Initial release.	N/A

Microsemi Corporate Headquarters
One Enterprise, Aliso Viejo,
CA 92656 USA
Within the USA: +1 (800) 713-4113
Outside the USA: +1 (949) 380-6100
Sales: +1 (949) 380-6136
Fax: +1 (949) 215-4996
E-mail: sales.support@microsemi.com
www.microsemi.com

About Microsemi

Microsemi Corporation (Nasdaq: MSCC) offers a comprehensive portfolio of semiconductor and system solutions for aerospace & defense, communications, data center and industrial markets. Products include high-performance and radiation-hardened analog mixed-signal integrated circuits, FPGAs, SoCs and ASICs; power management products; timing and synchronization devices and precise time solutions, setting the world's standard for time; voice processing devices; RF solutions; discrete components; enterprise storage and communication solutions, security technologies and scalable anti-tamper products; Ethernet solutions; Power-over-Ethernet ICs and midspans; as well as custom design capabilities and services. Microsemi is headquartered in Aliso Viejo, Calif., and has approximately 4,800 employees globally. Learn more at www.microsemi.com.