XAPP463 (v1.1.2) July 23, 2003

Using Block RAM in Spartan-3 FPGAs

Summary

For applications requiring large, on-chip memories, Spartan™-3 FPGAs provides plentiful, efficient SelectRAM™ memory blocks. Using various configuration options, SelectRAM blocks create RAM, ROM, FIFOs, large look-up tables, data width converters, circular buffers, and shift registers, each supporting various data widths and depths. This application note describes the features and capabilities of block SelectRAM and illustrates how to specify the various options using the Xilinx CORE Generator™ system or via VHDL or Verilog instantiation. Various non-obvious block RAM applications are discussed with references to additional tools, application notes, and documentation.

Introduction

All Spartan-3 devices feature multiple block RAM memories, organized in columns. The total amount of block RAM memory depends on the size of the Spartan-3 device, as shown in Table 1.

Table 1: **Block RAM Available in Spartan-3 Devices**

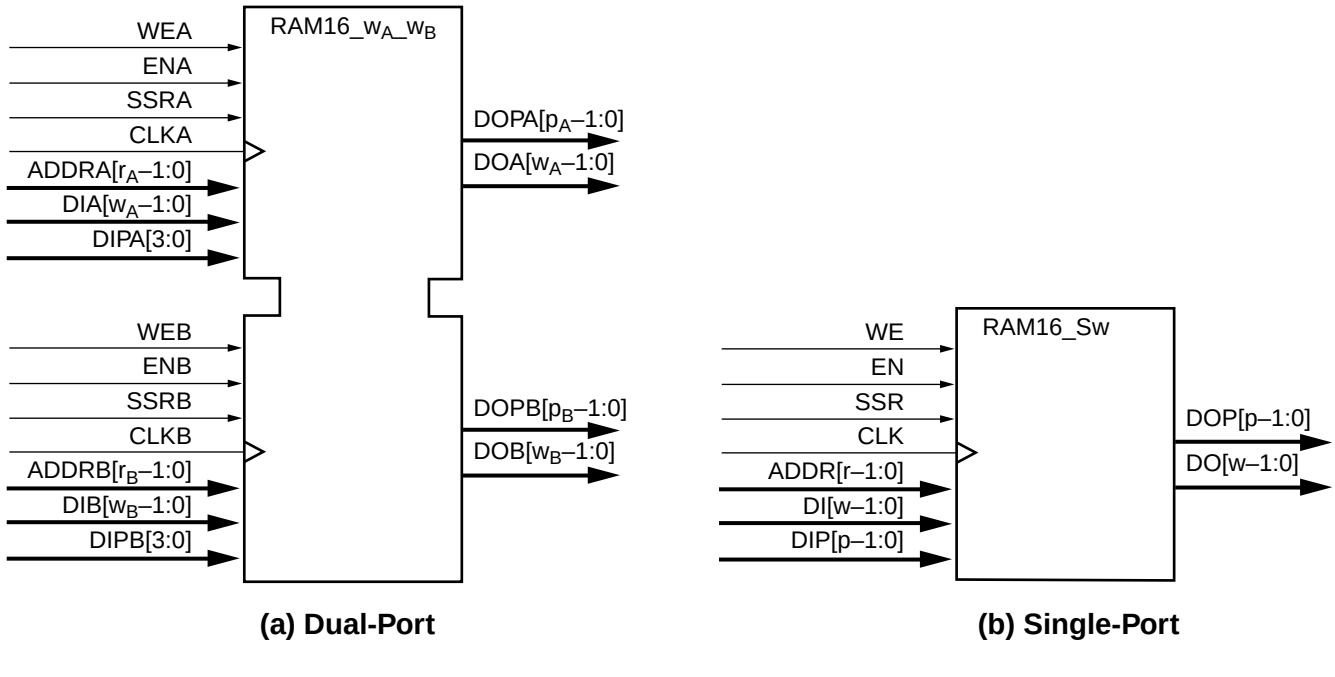
Spartan-3 Device	RAM Columns	RAM Blocks Per Column	Total RAM Blocks	Total RAM Bits	Total RAM Kbits
XC3S50	1	4	4	73,728	72K
XC3S200	2	6	12	221,184	216K
XC3S400	2	8	16	294,912	288K
XC3S1000	2	12	24	442,368	432K
XC3S1500	2	16	32	589,824	576K
XC3S2000	2	20	40	737,280	720K
XC3S4000	4	24	96	1,769,472	1,728K
XC3S5000	4	26	104	1,916,928	1,872K

Notes:
1. 1Kbit = 1,024 bits, per memory conventions.

Each block RAM contains 18,432 bits of fast static RAM, 16K bits of which is allocated to data storage and, in some memory configurations, an additional 2K bits allocated to parity or additional "plus" data bits. Physically, the block RAM memory has two completely independent access ports, labeled Port A and Port B. The structure is fully symmetrical, and both ports are interchangeable and both ports support data read and write operations. Each memory port is synchronous, with its own clock, clock enable, and write enable. Read operations are also synchronous and require a clock edge and clock enable.

Though physically a dual-port memory, block RAM simulates single-port memory in an application, as shown in Figure 1. Furthermore, each block memory supports multiple configurations or aspect ratios. Table 2 summarizes the essential SelectRAM features.

Cascade multiple block RAMs to create deeper and wider memory organizations, with a minimal timing penalty incurred through specialized routing resources.



Notes:
1. w_A and w_B are integers representing the total data path width (i.e., data bits plus parity bits) at ports A and B, respectively.
2. p_A and p_B are integers that indicate the number of data path lines serving as parity bits.
3. r_A and r_B are integers representing the address bus width at ports A and B, respectively.
4. The control signals CLK, WE, EN, and SSR on both ports have the option of inverted polarity.

Figure 1: **SelectRAM 18K Blocks Perform as Dual-Port (a) and Single-Port (b) Memory**

Table 2: **SelectRAM 18K Block Memory Features and Applications**

Total RAM bits, including parity	18,432 (16K data + 2K parity)
Memory Organizations	16Kx1 8Kx2 4Kx4 2Kx8 (no parity) 2Kx9 (x8 + parity) 1Kx16 (no parity) 1Kx18 (x16 + 2 parity) 512x32 (no parity) 512x36 (x32 + 4 parity) 256x72 (single-port only)
Parity	Available and optional only for organizations greater than byte-wide. Parity bits optionally available as extra data bits.
Performance	200 MHz (estimated)

Table 2: **SelectRAM 18K Block Memory Features and Applications *(Continued)***

Timing Interface	Simple synchronous interface. Similar to reading and writing from a register with a setup time for write operations and clock-to-output delay for read operations.
Single-Port	Yes
True Dual-Port	Yes
ROM, Initial RAM Contents	Yes
Mixed Data Port Widths	Yes
Power-Up Condition	User-defined data, defaults to zero
Potential Applications	Local data storage, FIFOs, elastic stores, register files, buffers, stacks, circular buffers, shift registers, delay lines, waveform storage and generation, direct digital synthesis, CAMs, associative memories, function tables, function generators, wide logic functions, code converters, encoders, decoders, counters, state machines, microsequencers, program storage for embedded processor(s)

The Xilinx CORE Generator system supports various modules containing block RAM for Spartan-3 devices including:

- Embedded dual- or single-port RAM modules
- ROM modules
- Synchronous and asynchronous FIFO modules
- Content-Addressable Memory (CAM) modules

Furthermore, block RAM can be instantiated in any synthesis-based design using the appropriate “RAMB16” module from the Xilinx design library.

This application note describes the signals and attributes of the Spartan-3 block RAM feature, including details on the various attributes and applications for block RAM.

Block RAM Location and Surrounding Neighborhood

As mentioned previously, block RAM is organized in columns. Figure 2 shows the Block RAM column arrangement for the XC3S200. The XC3S50 has a single column of block RAM, located two CLB columns from the left edge of the device. Spartan-3 devices larger than the XC3S50 have two columns of block RAM, adjacent to the left and right edges of the die, located two columns of CLBs from the I/Os at the edge. In addition to the block RAM columns at the edge, the XC3S4000 and XC3S5000 have two additional columns—a total of four columns—nearly equally distributed between the two edge columns. Table 1 describes the number of columns and the total amount of block RAM on a specific device. The edge columns make block RAM particularly useful in buffering or resynchronizing buses entering or leaving the Spartan-3 device.

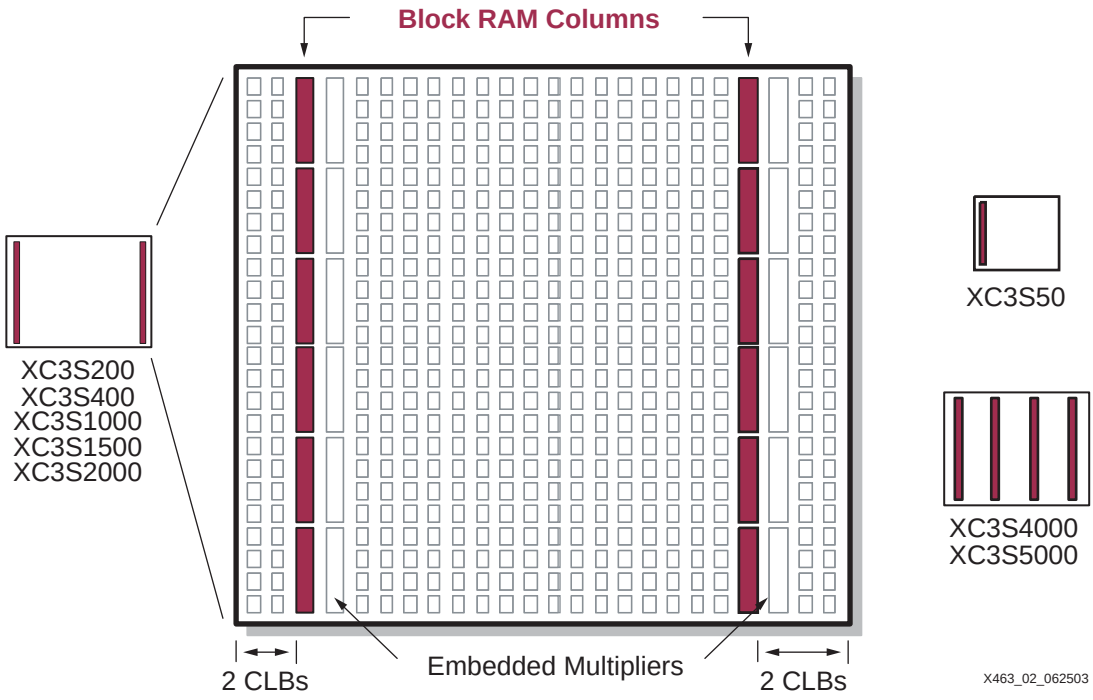


Figure 2: **Block RAMs Arranged in Columns with Detailed Floorplan of XC3S200**

Immediately adjacent to each block RAM is an embedded 18x18 hardware multiplier. Co-locating block RAM and the embedded multipliers improves the performance of some digital signal processing functions.

Special interconnect surrounding the block RAM provides efficient signal distribution for address and data. Furthermore, special provisions allow multiple block RAMs to be cascaded to create wider or deeper memories.

Data Flows

Spartan-3 block RAM is constructed of true dual-port memory and simultaneously supports all the data flows and operations shown in Figure 3. Both ports access the same set of memory bits but with two potentially different address schemes depending on the port's data width.

1. Port A behaves as an independent single-port RAM supporting simultaneous read and write operations using a single set of address lines.
2. Port B behaves as an independent single-port RAM supporting simultaneous read and write operations using a single set of address lines.
3. Port A is the write port with a separate write address and Port B is the read port with a separate read address. The data widths for Port A and Port B can be different also.
4. Port B is the write port with a separate write address and Port A is the read port with a separate read address. The data widths for Port B and Port A can be different also.

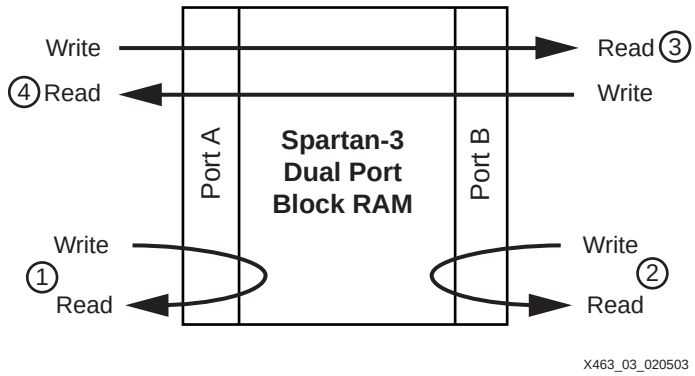


Figure 3: **Block RAM Support Single- and Dual-Port Data Transfers**

Signals

The signals connected to a block RAM primitive divide into four categories, as listed below. Table 3 lists the block RAM interface signals, the signals names for both single-port and dual-port memories, and signal direction.

1. Data Inputs and Outputs
2. Parity Inputs and Outputs, available when a data port is byte-wide or wider
3. Address inputs to select a specific memory location
4. Various control signals that manage read, write, or set/reset operations.

Table 3: **Block RAM Interface Signals**

Signal Description	Single Port	Dual Port		Direction
		Port A	Port B	
Data Input Bus	DI	DIA	DIB	Input
Parity Data Input Bus (available only for byte-wide and wider organizations)	DIP	DIPA	DIPB	Input
Data Output Bus	DO	DOA	DOB	Output
Parity Data Output (available only for byte-wide and wider organizations)	DOP	DOPA	DOPB	Output
Address Bus	ADDR	ADDRA	ADDRB	Input
Write Enable	WE	WEA	WEB	Input
Clock Enable	EN	ENA	ENB	Input
Synchronous Set/Reset	SSR	SSRA	SSRB	Input
Clock	CLK	CLKA	CLKB	Input

Data Inputs and Outputs

The total width of a port's data port includes both the data bus and the parity bus, when applicable, as shown in Figure 4. In the 512x36 organization, for example, the 36-bit data port width includes four parity bits as the more significant bits followed by the 32 data bits as the less significant bits.

The data and parity input and output signals are always buses; that is, in a 1-bit width configuration, the data input signal is DI[0] and the data output signal is DO[0].

Data Input Bus — DI[#:0] (DIA[#:0], DIB[#:0])

The Data Input bus is the source of data to be written into RAM.

Data at the DI input bus is written to the RAM location specified by the address input bus, ADDR, during a Low-to-High transition on the CLK input, when the clock enable EN and write enable WE inputs are High.

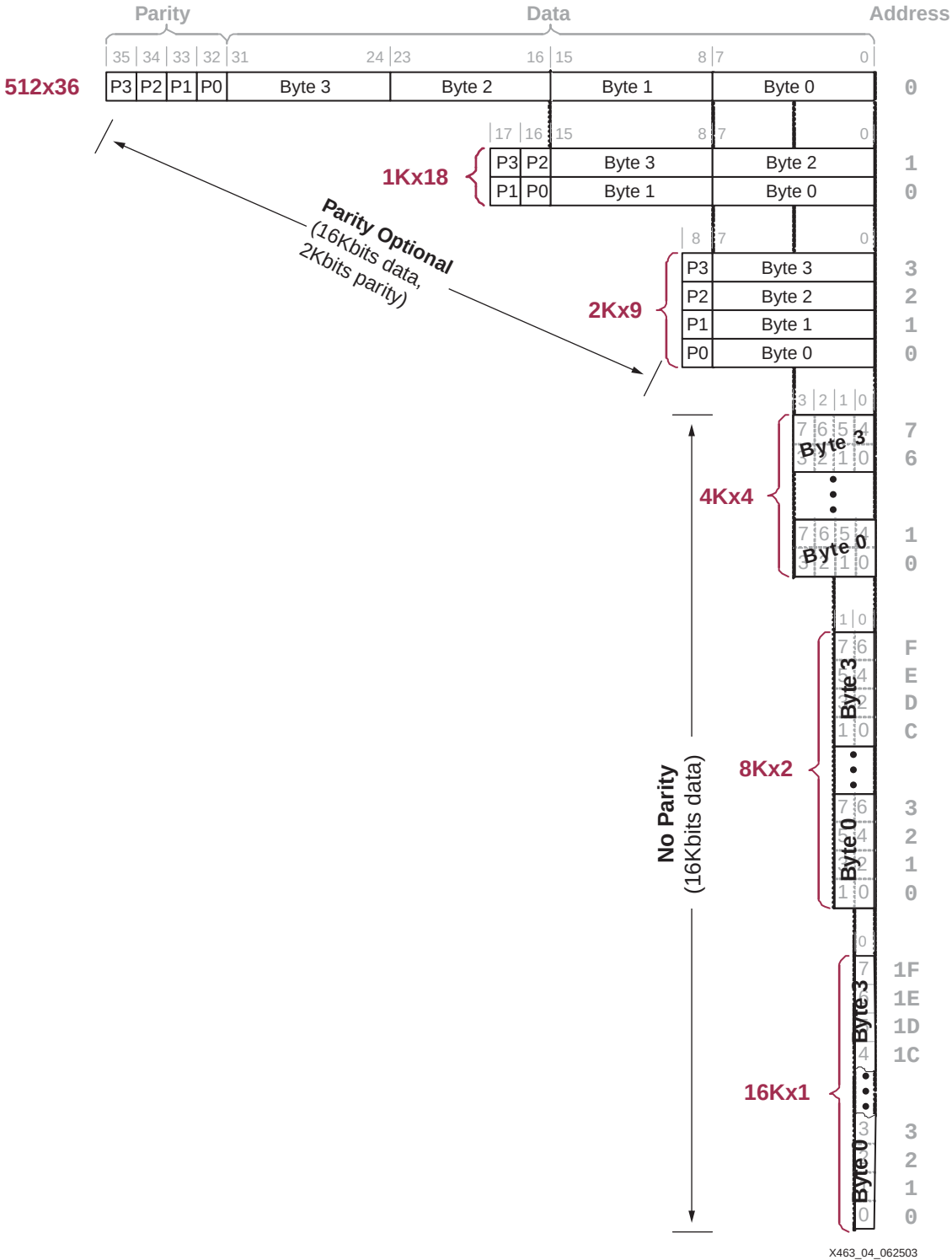


Figure 4: **Data Organization and Mapping Between Modes**

Data Output Bus — DO[#:0] (DOA[#:0], DOB[#:0])

The data output bus, DO, presents the contents of memory cells referenced by the address bus, ADDR, at the active clock edge during a read operation. During a simultaneous write operation, the behavior of the data output latches is controlled by the WRITE_MODE attribute (see **Read Behavior During Simultaneous Write — WRITE_MODE**, page 14).

Parity Inputs and Outputs

Parity is only supported for data paths byte wide and wider.

Although referred to herein as “parity” bits, the parity inputs and outputs have no special functionality and can be used as additional data bits. For example, the parity bits could be used to hold additional information about a data word, tagging the data as code or data, positive or negative values, old or new data, *etc*.

Block RAM does not contain any special circuitry for generating or checking parity. These functions, if required by the application, are created using CLB logic resources.

Data Input Parity Bus — DIP[#:0] (DIPA[#:0], DIPB[#:0])

Data at the DIP input bus is written to the RAM location specified by the address input bus, ADDR, during a Low-to-High transition on the CLK input, when the clock enable EN and write enable WE inputs are High.

Data Output Parity Bus — DOP[#:0] (DOPA[#:0], DOPB[#:0])

The data output bus, DOP, presents the contents of memory cells referenced by the address bus, ADDR, at the active clock edge during a read operation. During a simultaneous write operation, the behavior of the data output latches is controlled by the WRITE_MODE attribute (see **Read Behavior During Simultaneous Write — WRITE_MODE**, page 14).

Address Input

As dual-port RAM, both ports operate independently while accessing the same set of 18K-bit memory cells.

Address Bus — ADDR[#:0] (ADDRA[#:0], ADDRB[#:0])

The address bus selects the memory cells for read or write operations. The width of the address bus input determines the required address bus width, as shown in Table 5.

Control Inputs

Clock — CLK (CLKA, CLKB)

Each port is fully synchronous with independent clock pins. All port input pins have setup time referenced to the port CLK pin. The data bus has a clock-to-out time referenced to the CLK pin. Clock polarity is configurable and is rising edge triggered by default.

With default polarity, a Low-to-High transition on the clock (CLK) input controls read, write, and reset operations.

Enable — EN (ENA, ENB)

The enable input, EN, controls read, write, and set/reset operations. When EN is Low, no data is written and the outputs DO and DOP retain the last state. The polarity of EN is configurable and is active High by default.

When EN is asserted, minus an active synchronous set/reset input or write enable input, block RAM always reads the memory location specified by the address bus, ADDR, at the rising clock edge.

Write Enable — WE (WEA, WEB)

The write enable input, WE, controls when data is written to RAM. When both EN and WE are asserted at the rising clock edge, the value on the data and parity input buses is written to memory location selected by the address bus.

The data output latches are loaded or not loaded according to the WRITE_MODE attribute.

The polarity of WE is configurable and is active High by default.

Synchronous Set/Reset — SSR (SSRA, SSRB)

The synchronous set/reset input, SSR, forces the data output latches to value specified by the SRVAL attribute. When SSR and the enable signal, EN, are High, the data output latches for the DO and DOP outputs are synchronously set to a '0' or '1' according to the SRVAL parameter.

A Synchronous Set/Reset operation does not affect RAM memory cells and does not disturb write operations on the other port.

The polarity of SSR is configurable and is active High by default.

Global Set/Reset — GSR

The global set/reset signal, GSR, is asserted automatically and momentarily at the end of device configuration. By instantiating the STARTUP primitive, the logic application can also assert GSR to restore the initial Spartan-3 state at any time. The GSR signal initializes the output latches to the INIT value. A GSR signal has no impact on internal memory contents.

Because GSR is a global signal and automatically connected throughout the device, the block RAM primitive does not have a GSR input pin.

Inverting Control Pins

For each port, the four control pins—CLK, EN, WE, and SSR—each have an individual inversion option. Any control signal can be configured as active High or Low, and the clock can be active on a rising or falling edge without consuming additional logic resources.

Unused Inputs

Tie any unused data or address inputs to logic '1'. Connecting the unused inputs High saves logic and routing resources compared to connecting the inputs Low.

Attributes

A block RAM has a number of attributes that control its behavior as shown in Table 4 for VHDL and Verilog. The CORE Generator system uses slightly different values, as described below.

Table 4: **Block RAM Attributes and VHDL/Verilog Attribute Names**

Function	VHDL or Verilog Attribute	Default Value
Number of Ports	Defined by instantiating the appropriate RAMB16 primitive	N/A
Memory Organization	Defined by instantiating the appropriate RAMB16 primitive	N/A
Initial Content for Data Memory, Loaded during Configuration	INIT_xx	Initialized to zero
Initial Content for Parity Memory, Loaded during Configuration	INITP_xx	Initialized to zero
Data Output Latch Initialization	INIT (single-port) INIT_A, INIT_B (dual-port)	Initialized to zero

Table 4: **Block RAM Attributes and VHDL/Verilog Attribute Names *(Continued)***

Function	VHDL or Verilog Attribute	Default Value
Data Output Latch Synchronous Set/Reset Value	SRVAL (single-port) SRVAL_A, SRVAL_B (dual-port)	Reset to zero
Data Output Latch Behavior during Write	WRITE_MODE	WRITE_FIRST
Block RAM Location	LOC	N/A

Number of Ports

Although physically dual-port memory, each block RAM performs as either single-port or dual-port memory. The method to specify the number of ports depends on the design entry tool.

CORE Generator System

As shown in Figure 5, the Xilinx CORE Generator system provides module generators for various types of memory blocks. Choose single- or dual-port block memories, or use the higher-level functions to create FIFOs, content-addressable memories (CAMs), and so forth.

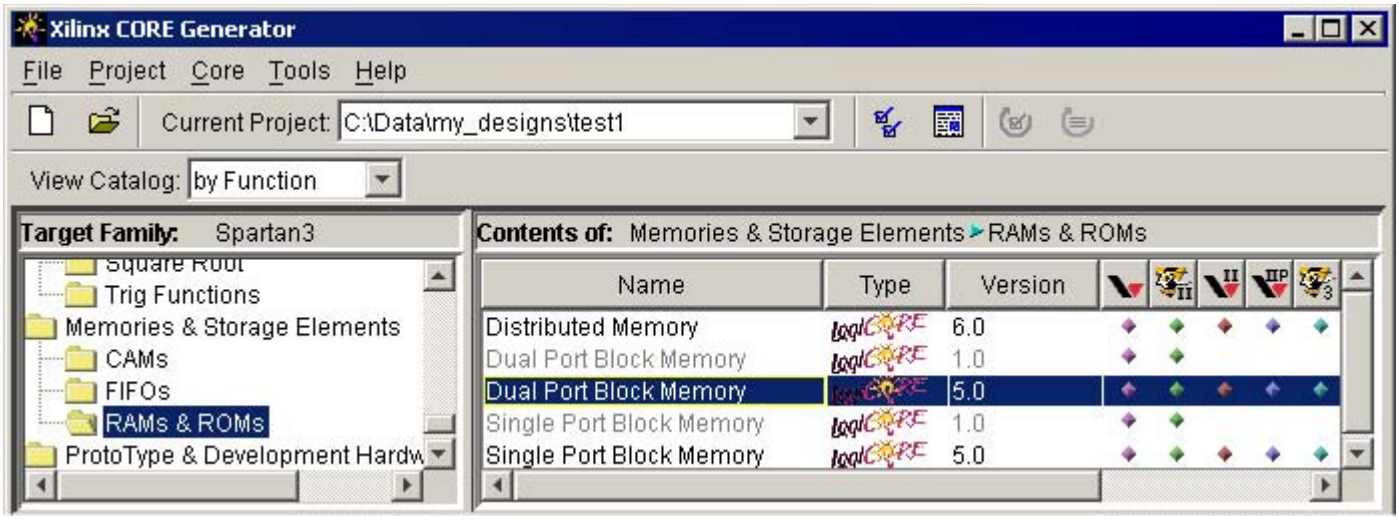


Figure 5: **Selecting a Block RAM Function in CORE Generator System**

VHDL or Verilog Instantiation

The Xilinx design libraries contain single- and dual-port memory primitives similar to those shown in Figure 1. Select among the various primitives to choose single- or dual-port memory, as well as the memory organization or aspect ratio of the memory. See Table 5 and Table 6 for single-port and dual-port block RAM primitives, respectively.

Memory Organization/Aspect Ratio

The data organization or aspect ratio of a RAM block is configurable, as shown in Table 5. If the data path is byte-wide or wider, then the block RAM also provides additional bits to support parity for each byte. Consequently, a 1Kx18 memory organization is 18 bits wide with 16 bits (two bytes) allocated to data plus two parity bits, one for each byte. Also, the physical amount of memory accessible from a port depends on the memory organization. For memories byte-wide and wider, there are 18K memory bits accessible. For narrower memories, only 16K bits are accessible due to the lack of parity bits in these organizations. Essentially, 16K bits are allocated to data, 2K bits to parity on the 18K-bit block RAM. See Figure 4 for details on data mapping for and between each memory organization.

Table 5: **Block RAM Data Organizations/Aspect Ratios**

Organization	Memory Depth	Data Width	Parity Width	DI/DO	DIP/DOP	ADDR	Single-Port Primitive	Total RAM Kbits
512x36	512	32	4	(31:0)	(3:0)	(8:0)	**RAMB16_S36**	18K
1Kx18	1024	16	2	(15:0)	(1:0)	(9:0)	**RAMB16_S18**	18K
2Kx9	2048	8	1	(7:0)	(0:0)	(10:0)	**RAMB16_S9**	18K
4Kx4	4096	4	-	(3:0)	-	(11:0)	**RAMB16_S4**	16K
8Kx2	8192	2	-	(1:0)	-	(12:0)	**RAMB16_S2**	16K
16Kx1	16384	1	-	(0:0)	-	(13:0)	**RAMB16_S1**	16K

CORE Generator System — Memory Size

The CORE Generator system creates a wide variety of memories with very flexible aspect ratios. Unlike the actual block RAM primitive, the CORE generator system does not differentiate between data and parity bits and considers all bits data bits. For dual-port memories, each port can have different organizations or aspect ratios.

Within the CORE Generator system, locate the Memory Size group and enter the desired memory organization, as shown in Figure 6.

Figure 6: **Selecting Memory Width and Depth in CORE Generator System**

VHDL or Verilog Instantiation

The aspect ratio is defined at design time by specifying or instantiating the appropriate SelectRAM component. Table 5 indicates the SelectRAM component for single-port RAM. For single-port RAM, the proper component name is `RAMB16_S`***n***, where ***n*** is the data path width including both the data bits plus parity bits. For example, a 1Kx18 single-port RAM uses component `RAMB16_S18`. In this example, ***n***=18 because there are 16 data bits plus 2 parity bits.

Selecting a dual-port memory is slightly more complex because the two memory ports may have different aspect ratios. For dual-port RAM, the proper component name is `RAMB16_S`***m***`_S`***m***, where ***m*** is the data path width for Port A and ***n*** is the width for Port B. For example, using the suffix shown in Table 6, if Port A is organized a 2Kx9 and Port B is organized as 1Kx18, then the proper dual-port RAM component is `RAMB16_S9_S18`. In this example, ***m***=9 and ***n***=18.

Table 6: **Dual-Port RAM Component Suffix Appended to "RAMB16"**

		Port A					
		16Kx1	**8Kx2**	**4Kx4**	**2Kx9**	**1Kx18**	**512x36**
Port B	16Kx1	**_S1_S1**					
	8Kx2	**_S1_S2**	**_S2_S2**				
	4Kx4	**_S1_S4**	**_S2_S4**	**_S4_S4**			
	2Kx9	**_S1_S9**	**_S2_S9**	**_S4_S9**	**_S9_S9**		
	1Kx18	**_S1_S18**	**_S2_S18**	**_S4_S18**	**_S9_S18**	**_S18_S18**	
	512x36	**_S1_S36**	**_S2_S36**	**_S4_S36**	**_S9_S36**	**_S18_S36**	**_S36_S36**

Address and Data Mapping Between Two Ports

In dual-port mode, both ports access the same set of memory cells. However, both ports may have the same or different memory organization or aspect ratio. Figure 4 shows how the same data set may appear with different aspect ratios.

There are extra bits available to store parity for memory organizations that are byte-wide or wider. The extra parity bits are designed to be associated with a particular byte and these parity bits appear as the more-significant bits on the data port. For example, if a x36 data word (32 data, 4 parity) is addressed as two x18 halfwords (16 data, 2 parity), the parity bits associated with each data byte are mapped within the block RAM to appropriate parity bits. The same effect happens when the x36 data word is mapped as four x9 words. The extra parity bits are not available if the data port is configured as x4, x2, or x1.

The following formulas provide the starting and ending address for data when the two ports have different memory organizations. Find the starting and ending address for Port X given the address and port width of Port Y and the port width of Port X.

$$\text{START\_ADDRESS}_X = \text{INTEGER}\left(\frac{\text{ADDRESS}_Y \bullet \text{WIDTH}_Y}{\text{WIDTH}_X}\right)$$

$$\text{END\_ADDRESS}_X = \text{INTEGER}\left(\frac{((\text{ADDRESS}_Y + 1) \bullet \text{WIDTH}_Y) - 1}{\text{WIDTH}_X}\right)$$

If, due the memory organization, one port includes parity bits and the other does not, then the above equations are invalid and the values for width should only include the data bits. The parity bits are not available on any port that is less than 8 bits wide.

Content Initialization

By default, block RAM memory is initialized with all zeros during the device configuration sequence. However, the contents can also be initialized with user-defined data. Furthermore, the RAM contents are protected against spurious writes during configuration.

CORE Generator System — Load Init File

To specify the initial RAM contents for a CORE Generator block RAM function, create a coefficients (.coe) file. A simple example of a coefficients file appears in Figure 7. At a minimum, define the radix for the initialization data—*i.e.*, base 2, 10, or 16—and then specify the RAM contents starting with the data at location 0, followed by data at subsequent locations.

```
memory_initialization_radix=16;
memory_initialization_vector= 80, 0F, 00, 0B, 00, 0C, …, 81;
```

Figure 7: **A Simple Coefficients File (.coe) Example**

To include the coefficients file, locate the appropriate section in the CORE Generator wizard and check **Load Init File**, as shown in Figure 8. Then, click **Load File** and select the coefficients file.

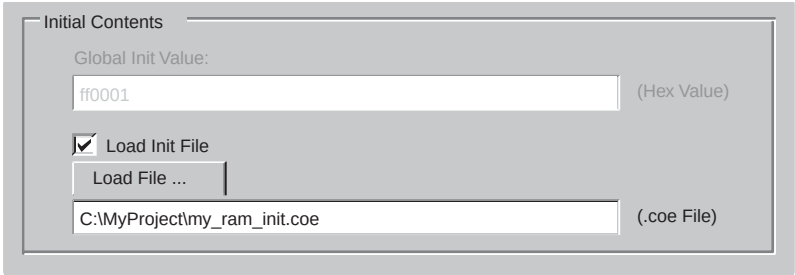


Figure 8: **Specifying Initial RAM Contents in CORE Generator System**

VHDL or Verilog Instantiation — INIT_xx, INITP_xx

For VHDL and Verilog instantiation, there are two different types of initialization attributes. The **INIT_xx** attributes define the initial contents of the data memory locations. The **INITP_xx** attributes define the initial contents of the parity memory locations.

The **INIT_xx** attributes on the instantiated primitive define the initial memory contents. There are 64 initialization attributes, named **INIT_00** through **INIT_3F**. Each **INIT_xx** attribute is a 64-digit (256-bit) hex-encoded bit vector. The memory contents can be partially initialized and any unspecified locations are automatically completed with zeros.

The following formula defines the bit positions for each **INIT_xx** attribute.

Given yy = convert_hex_to_decimal(xx), **INIT_xx** corresponds to the following memory cells.

- Starting Location: [(yy + 1) * 256] –1
- End Location: (yy) * 256

For example, for the attribute INIT_1F, the conversion is as follows:

- yy = convert_hex_to_decimal(0x1F) = 31
- Starting Location: [(31+1) * 256] –1 = 8191
- End Location: 31 * 256 = 7936

Table 7: **VHDL/Verilog RAM Initialization Attributes for Block RAM**

Attribute	From	To
INIT_00	255	0
INIT_01	511	256
INIT_02	767	512
…	…	…
INIT_3F	16383	16128

The **INITP_xx** attributes define the initial contents of the memory cells corresponding to parity bits, *i.e.*, those bits that connect to the DIP/DOP buses. By default these memory cells are also initialized to all zeros.

The eight initialization attributes from **INITP_00** through **INITP_07** represent the memory contents of parity bits. Each **INITP_xx** is a 64-digit (256-bit) hex-encoded bit vector and behaves like an **INIT_xx** attribute. The same formula calculates the bit positions initialized by a particular **INITP_xx** attribute.

Data Output Latch Initialization

The block RAM output latches can be initialized to a user-specified value immediately after configuration or whenever the global set/reset signal, GSR, is asserted. For dual-port memories, there is a separate initialization value for each port.

If no value is specified, the output latch is initialized to zero.

CORE Generator System — Global Init Value

Figure 9 describes how to specify the initial value for data output latches in the CORE Generator system. The value, specified in hexadecimal, should include one bit per the specified data width. For dual-port memories, there is a separate initialization value for each port.

Figure 9: **Specifying Initial Value for Block RAM Data Output Latches**

VHDL or Verilog Instantiation — INIT (INIT_A and INIT_B)

For VHDL or Verilog, the INIT attribute (or INIT_A and INIT_B for dual-port memories) defines the output latch value after configuration. The INIT (or INIT_A and INIT_B) attribute specifies the initial value for the data and, if applicable, the parity bits. Figure 4 shows the expected bit format for each memory organization with parity bits—if applicable—as the more significant bits followed by the data bits. For example, the initialization value for a 2Kx9 memory would be nine bits wide and would include one parity bit followed by eight data bits. These attributes are hex-encoded bit vectors and the default value is 0.

Data Output Latch Synchronous Set/Reset Value

When the synchronous set/reset input, SSR, is asserted, the data output latches are set or reset according to the set/reset value attribute. For dual-port memories, there is a separate initialization value for each port.

If no value is specified, the output latch is reset to zero during a valid Synchronous Set/Reset operation.

CORE Generator System — Init Value (SINIT)

Figure 10 describes how to specify the synchronous set/reset value for data output latches in the CORE Generator system. Check the **SINIT pin** and then specify the synchronous set/reset value in hexadecimal, with one bit per the specified data width. For dual-port memories, there is a separate value for each port.

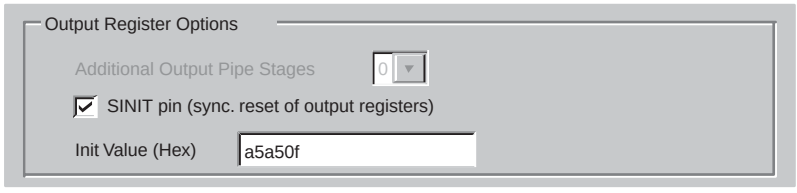


Figure 10: **Specifying the Output Data Latch Set/Reset Value**

VHDL or Verilog Instantiation — SRVAL (SRVAL_A and SRVAL_B)

For VHDL or Verilog, the SRVAL attribute (or SRVAL_A and SRVAL_B for dual-port memories) defines the output latch value after configuration. The SRVAL (or SRVAL_A and SRVAL_B) attribute specifies the initial value for the data and, if applicable, the parity bits. Figure 4 shows the expected bit format for each memory organization with parity bits—if applicable—as the more significant bits followed by the data bits. These attributes are hex-encoded bit vectors and the default value is 0.

Read Behavior During Simultaneous Write — WRITE_MODE

To maximize data throughput and utilization of the dual-port memory at each clock edge, block RAM memory supports one of three write modes for each memory port. These different modes determine which data is available on the output latches after a valid write clock edge to the same port. The default mode, WRITE_FIRST, provides backwards compatibility with the older Virtex™/E and Spartan-IIE FPGA architectures and is also the default behavior for Virtex-II/Pro devices. However, READ_FIRST mode is the most useful as it increases the efficiency of block RAM memory at each clock cycle, allowing designs to use maximum bandwidth. In READ_FIRST mode, a memory port supports simultaneous read and write operations to the same address on the same clock edge, free of any timing complications.

Table 8 outlines how the WRITE_MODE setting affects the output data latches on the same port, and how it affects the output latches on the opposite port during a simultaneous access to the same address.

Table 8: **WRITE_MODE Affects Data Output Latches During Write Operations**

Write Mode	Effect on Same Port	Effect on Opposite Port (dual-port mode only, same address)
WRITE_FIRST Read After Write (Default)	Data on DI, DIP inputs written into specified RAM location and simultaneously appears on DO, DOP outputs.	Invalidates data on DO, DOP outputs.
READ_FIRST Read Before Write (Recommended)	Data from specified RAM location appears on DO, DOP outputs. Data on DI, DIP inputs written into specified location.	Data from specified RAM location appears on DO, DOP outputs.
NO_CHANGE No Read on Write	Data on DO, DOP outputs remains unchanged. Data on DI, DIP inputs written into specified location.	Invalidates data on DO, DOP outputs.

Mode selection is set by configuration. One of these three modes is set individually for each port by an attribute. The default mode is WRITE_FIRST.

WRITE_FIRST or Transparent Mode (Default)

The WRITE_FIRST mode is the default operating mode for backward compatibility reasons. For new designs, READ_FIRST mode is recommended.

In this mode, the input data is written into the addressed RAM location memory and simultaneously stored in the data output latches, resulting in a transparent write operation, as shown in Figure 11. The WRITE_FIRST mode provides backwards compatibility with the 4K-bit blocks RAMs on Virtex/E and Spartan-II/E FPGAs and is also the default mode for Virtex-II/Pro block RAMs.

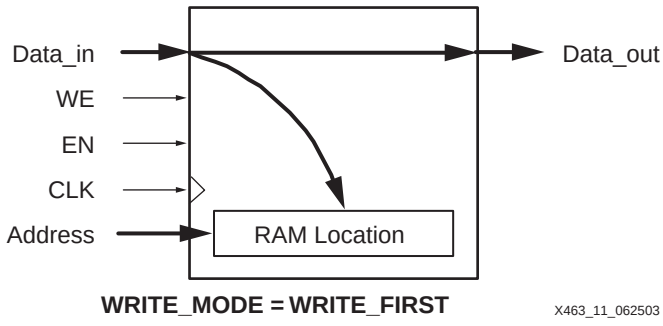


Figure 11: **Data Flow during a WRITE_FIRST Write Operation**

Figure 12 demonstrates that a valid write operation during a valid read operation results in the write data appearing on the data output.

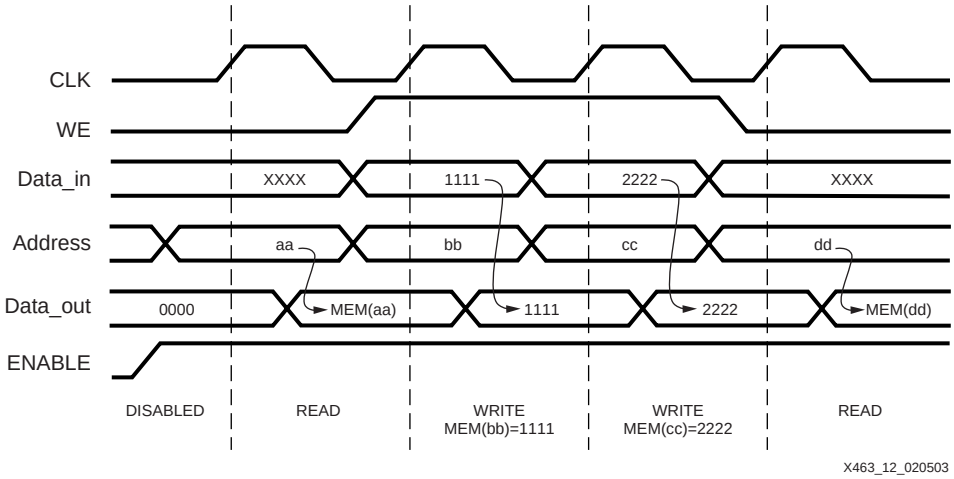


Figure 12: **WRITE_FIRST Mode Waveforms**

READ_FIRST or Read-Before-Write Mode

In READ_FIRST mode, data previously stored at the write address appears on the output latches, while the new input data is stored in memory, resulting in a read-before-write operation shown in Figure 13. The older RAM data appears on the data output while the new RAM data is stored in the specified RAM location. READ_FIRST mode is the recommended operating mode.

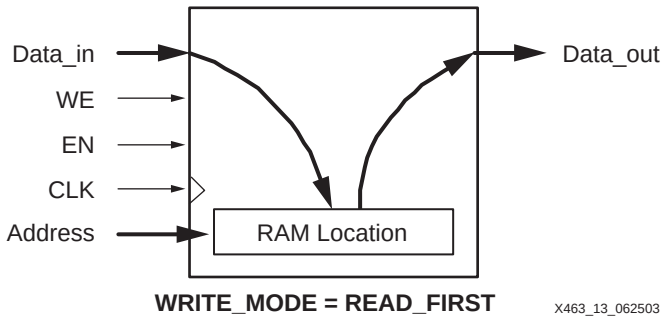


Figure 13: **Data Flow during a READ_FIRST Write Operation**

Figure 14 demonstrates that the older RAM data always appears on the data output, regardless of a simultaneous write operation.

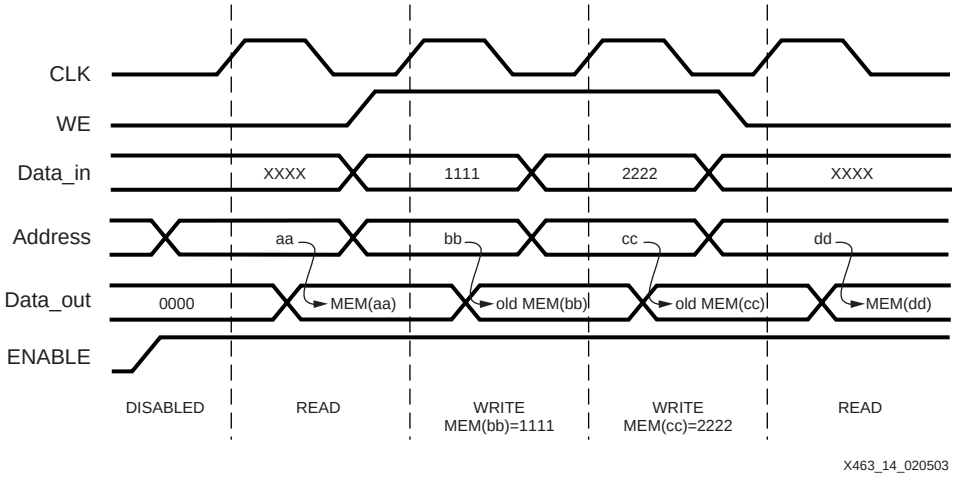


Figure 14: **READ_FIRST Mode Waveforms**

This mode is particularly useful for building circular buffers and large, block-RAM-based shift registers. Similarly, this mode is useful when storing FIR filter taps in digital signal processing applications. Old data is copied out from RAM while new data is written into RAM.

NO_CHANGE Mode

In NO_CHANGE mode, the output latches are disabled and remain unchanged during a simultaneous write operation, as shown in Figure 15. This behavior mimics that of simple synchronous memory where a memory location is either read or written during a clock cycle, but not both.

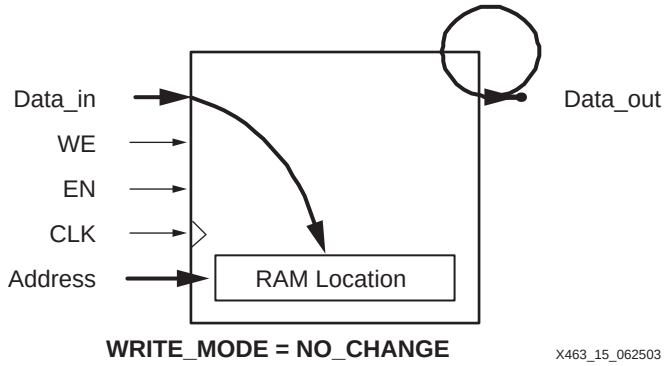


Figure 15: **Data Flow during a NO_CHANGE Write Operation**

The NO_CHANGE mode is useful in a variety of applications, including those where the block RAM contains waveforms, function tables, coefficients, and so forth. The memory can be updated without affecting the memory output.

Figure 16 shows that the data output retains the last read data if there is a simultaneous write operation on the same port.

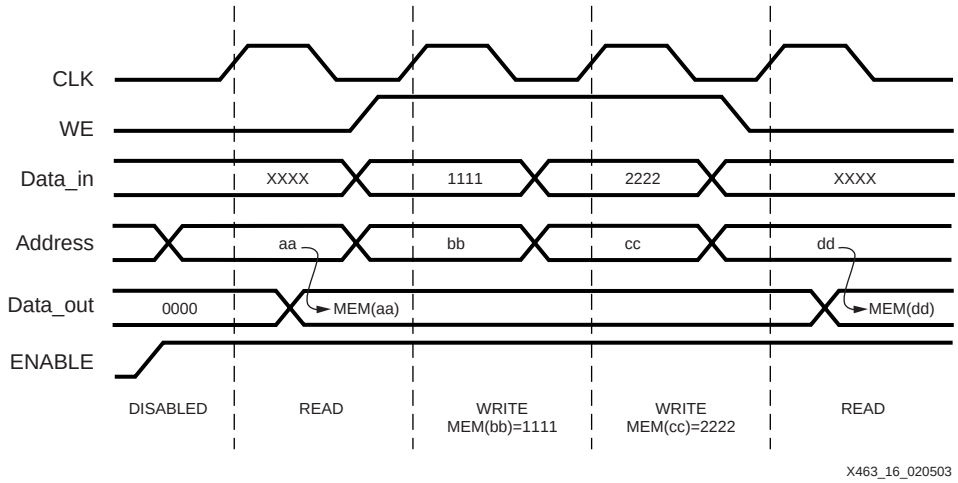


Figure 16: **NO_CHANGE Mode Waveforms**

CORE Generator System — Write Mode

To specify the WRITE_MODE in the CORE Generator system, locate the settings for Write Mode as shown in Figure 17. Select between Read After Write (WRITE_FIRST), Read Before Write (READ_FIRST) or No Read On Write (NO_CHANGE).

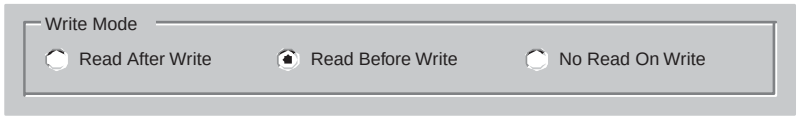


Figure 17: **Selecting the Write Mode in CORE Generator System**

VHDL or Verilog Instantiation — WRITE_MODE

When instantiating block RAM, specify the write mode via the WRITE_MODE attribute. Acceptable values include WRITE_FIRST, READ_FIRST, and NO_CHANGE, as demonstrated in the examples in the appendices.

Location Constraints (LOC)

In general, it is best to allow the Xilinx ISE software to assign a block RAM location. However, block RAMs can be constrained to specific locations on a Spartan-3 device using an attached LOC property. Block RAM placement locations are device specific and differ from the convention used for naming CLB locations, allowing LOC properties to transfer easily from array to array.

The LOC properties use the following form:

```
LOC = RAMB16_X#Y#
```

The **RAMB16_X0Y0** is the lower-left block RAM location on the device, as shown in Figure 18. The upper-right block RAM location depends on **n**, the number of block RAM columns, and **m**, the number of block RAM rows, as provided in Table 1.

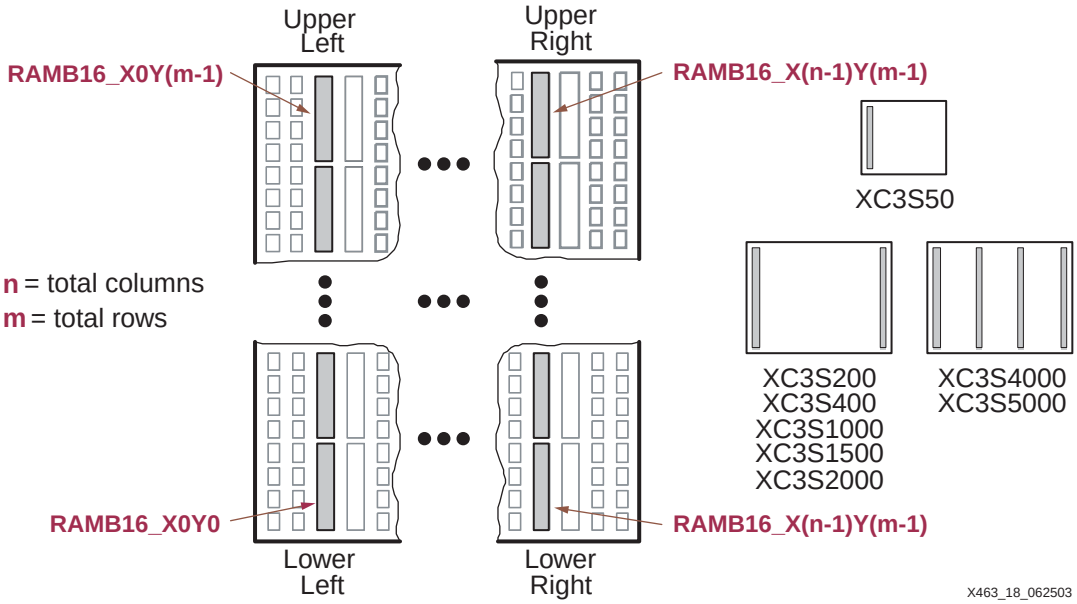


Figure 18: **Block RAM LOC Coordinates**

Location attributes cannot be specified directly in the CORE Generator system. However, location constraints can be added to VHDL or Verilog instantiations.

Block RAM Operation

Table 9 describes the behavior of block RAM and assumes that all control signals use their default, active-High behavior. However, the control signals can be inverted in the design if necessary. The table and following text describes the behavior for a single memory port. In dual-port mode, both ports perform as independent single-port memories.

All read and write operations to block RAM are synchronous. All inputs have a set-up time relative to clock and all outputs have a clock-to-output time.

Table 9: **Block RAM Function Table**

Input Signals								Output Signals		RAM Contents	
GSR	**EN**	**SSR**	**WE**	**CLK**	**ADDR**	**DIP**	**DI**	**DOP**	**DO**	**Parity**	**Data**
Immediately After Configuration											
Loaded During Configuration								X	X	INITP_xx[2]	INIT_xx[2]
Global Set/Reset Immediately after Configuration											
1	X	X	X	X	X	X	X	INIT[3]	INIT	No Chg	No Chg
RAM Disabled											
0	0	X	X	X	X	X	X	No Chg	No Chg	No Chg	No Chg
Synchronous Set/Reset											
0	1	1	0	↑	X	X	X	SRVAL[4]	SRVAL	No Chg	No Chg
Synchronous Set/Reset during Write RAM											
0	1	1	1	↑	addr	pdata	Data	SRVAL	SRVAL	RAM(addr) ←pdata	RAM(addr) ← data

Table 9: **Block RAM Function Table *(Continued)***

Input Signals								Output Signals		RAM Contents	
GSR	**EN**	**SSR**	**WE**	**CLK**	**ADDR**	**DIP**	**DI**	**DOP**	**DO**	**Parity**	**Data**
Read RAM, no Write Operation											
0	1	0	0	↑	addr	X	X	RAM(pdata)	RAM(data)	No Chg	No Chg
Write RAM, Simultaneous Read Operation											
0	1	0	1	↑	addr	pdata	Data	**WRITE_MODE =** WRITE_FIRST[5] **(default)**			
								pdata	data	RAM(addr) ←pdata	RAM(addr) ← data
								WRITE_MODE = READ_FIRST[6] **(recommended)**			
								RAM(data)	RAM(data)	RAM(addr) ←pdata	RAM(addr) ←pdata
								WRITE_MODE = NO_CHANGE[7]			
								No Chg	No Chg	RAM(addr) ←pdata	RAM(addr) ←pdata

Notes:

1. No Chg = No Change, addr = address to RAM, data = RAM data, pdata = RAM parity data.
2. Refer to **Content Initialization**, page 11.
3. Refer to **Data Output Latch Initialization**, page 12.
4. Refer to **Data Output Latch Synchronous Set/Reset Value**, page 13.
5. Refer to **WRITE_FIRST or Transparent Mode (Default)**, page 14.
6. Refer to **READ_FIRST or Read-Before-Write Mode**, page 15.
7. Refer to **NO_CHANGE Mode**, page 16.

RAM Contents Initialized During Configuration

The initial RAM contents, if specified, are loaded during the Spartan-3 configuration process. If no contents are specified, the RAM cells are loaded with zero. The RAM contents are protected against spurious writes during configuration.

Global Set/Reset Initializes Data Output Latches Immediately After Configuration or Global Reset

Immediately following configuration, the Spartan-3 device begins its start-up procedure and asserts the global set/reset signal, GSR, to initialize the state of all flip-flops and registers. The initial contents of the block RAM output latches, INIT, are asynchronously loaded at this time. The GSR signal does not change or re-initialize the RAM contents.

Enable Input Activates or Disables RAM

If the block RAM is disabled—*i.e.*, EN is Low—then the block RAM retains its present state. The enable input must be High for any other operations to proceed.

Synchronous Set/Reset Initializes Data Output Latches

If the block RAM is enabled (EN is High) and the Synchronous Set/Reset signal is asserted High, then the data output latches are initialized at the next rising clock edge. The SRVAL attribute defines the synchronous set/reset state for the data output latches. This operation is different the operation caused by the global set/reset signal, GSR, immediately after configuration. The synchronous set/reset input affects the specific RAM block whereas the GSR signal affects the entire device.

Simultaneous Write and Synchronous Set/Reset Operations

If a simultaneous write operation occurs during the synchronous set/reset operation, then the data on the DI and DIP inputs is stored at the RAM location specified by the ADDR input. However, the data output latches are initialized to the SRVAL attribute value as described immediately above.

Read Operations Occur on Every Clock Edge When Enable is Asserted

Read operations are synchronous and require a clock edge and an asserted clock enable. The data output behavior depends on whether or not a simultaneous write operation occurs during the read cycle.

If no simultaneous write cycle occurs during a valid read cycle, then the read address is registered on the read port and the data stored in RAM at that address is simply loaded into the output latches after the RAM access interval passes.

However, if there is a simultaneous write cycle during the read cycle, then the output behavior depends on which of the three write modes is selected, as described immediately below.

Write Operations Always Have Simultaneous Read Operation, Data Output Latches Affected

During a Write operation, a simultaneous Read operation occurs. The WRITE_MODE attribute determines the behavior of the data output latches during the Write operation (refer to **Read Behavior During Simultaneous Write — WRITE_MODE**, page 14). By default, WRITE_MODE is WRITE_FIRST and the data output latches and the addressed RAM locations are updated with the input data during a simultaneous Write operation. When WRITE_MODE is READ_FIRST, the output latches are updated with the data previously stored in the addressed RAM location and the new data on the DI and DIP inputs is stored at the address RAM location. When WRITE_MODE is NO_CHANGE, the data output latches are unaffected by a simultaneous Write operation and retain their present state.

General Characteristics

- A write operation requires only one clock edge.
- A read operation requires only one clock edge.
- All inputs are registered with the port clock and have a setup-to-clock timing specification.
- All outputs have a read-through function or one of three read-during-write functions, depending on the state of the WE pin. The outputs relative to the port clock are available after the clock-to-out timing interval.
- Block RAM cells are true synchronous RAM memories and do not have a combinatorial path from the address to the output.
- The ports are completely independent of each other without arbitration. Each port has its own clocking, control, address, read/write functions, initialization, and data width.
- Output ports are latched with a self-timed circuit, guaranteeing glitch-free read operations. The state of the output port does not change until the port executes another read or write operation.

Functional Compatibility with Other Xilinx FPGA Families

The block RAM on Spartan-3 FPGAs is functionally identical to block RAM on the Xilinx Virtex-II/Pro FPGA families. Consequently, design tools that support Virtex-II and Virtex-II Pro block RAM also support with Spartan-3 FPGAs.

Dual-Port RAM Conflicts and Resolution

As a dual-port RAM, the block RAM allows both ports to simultaneously access the same memory cell. Potentially, conflicts arise under the following conditions.

1. If the clock inputs to the two ports are asynchronous, then conflicts occur if clock-to-clock setup time requirements are violated.
2. Both memory ports write different data to the same RAM location during a valid write cycle.
3. If a port uses WRITE_MODE=NO_CHANGE or WRITE_FIRST, a write to the port invalidates the read data output latches on the opposite port.

If Port A and Port B different memory organizations and consequently different widths, only the overlapping bits are invalid when conflicts occur.

Timing Violation Conflicts

When one port writes to a given memory cell, the other port must not address that memory cell—either for a write or a read operation—within the clock-to-clock setup window specified in the Spartan-3 data sheet. Figure 19 describes this situation where both ports operate from asynchronous clock inputs.

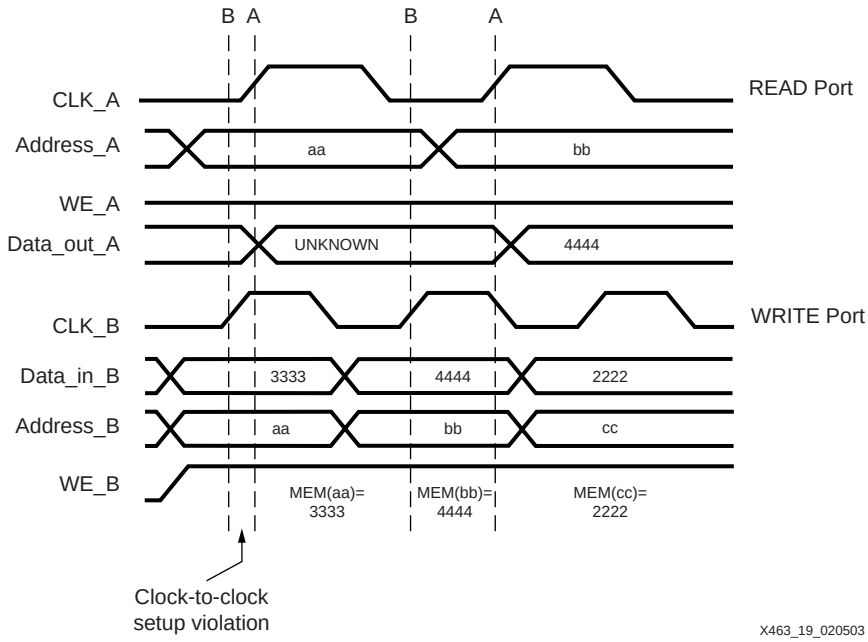


Figure 19: **Clock-to-Clock Timing Conflicts**

The first rising edge on CLK_A violates the clock-to-clock setup parameter, because it occurs too soon after the last CLK_B clock edge. The write operation on port B is valid because Data_in_B, Address_B, and WE_B all had sufficient set-up time before the rising edge on CLK_B. Unfortunately, the read operation on port A is invalid because it depends on the RAM contents being written to Address_B and the read clock, CLK_A, happened too soon after the write clock, CLK_B.

On the second rising edge of CLK_B, there is another valid write operation to port B. The memory location at address (bb) contains 4444. Data on the Data_out_A port is still invalid because there has not been another rising clock edge on CLK_A. The second rising edge of CLK_A reads the new data at the in location (bb), which now contains 4444. This time, the read operating is valid because there has been sufficient setup time between CLK_B and CLK_A.

Simultaneous Writes to Both Ports with Different Data Conflicts

If both ports write simultaneously into the same memory cell with different data, then the data stored in that cell becomes invalid, as outlined in Table 10.

Table 10: **RAM Conflicts During Simultaneous Writes to Same Address**

Input Signals								RAM Contents	
Port A				Port B					
WEA	CLKB	DIPA	DIA	WEB	CLKA	DIPB	DIB	Parity	Data
1	↑	DIPA	DIA	1	↑	DIPB	DIB	?	?

Notes:

1. ADDRA=ADDRB, ENA=1,ENB=1, DIPA ≠ DIPB, DIA ≠ DIB, ?=Unknown or invalid data.

Write Mode Conflicts on Output Latches

Potential conflicts occur when one port writes to memory and the opposite port reads from memory. Write operations always succeed and the write port's output data latches behave as described by the port's WRITE_MODE attribute. If the write port is configured with WRITE_MODE set to NO_CHANGE or WRITE_FIRST, then a write operation to the port invalidates the data output latches on the opposite port, as shown in Table 11.

Using the READ_FIRST mode does not cause conflicts on the opposite port.

Table 11: **Conflicts to Output Latches Based on WRITE_MODE**

Input Signals								Output Signals			
Port A				Port B				Port A		Port B	
WEA	CLKB	DIPA	DIA	WEB	CLKA	DIPB	DIB	DOPA	DOA	DOPB	DOB
WRITE_MODE_A=NO_CHANGE											
1	↑	DIPA	DIA	0	↑	DIPB	DIB	No Chg	No Chg	?	?
WRITE_MODE_B=NO_CHANGE											
0	↑	DIPA	DIA	1	↑	DIPB	DIB	?	?	No Chg	No Chg
WRITE_MODE_A=WRITE_FIRST											
1	↑	DIPA	DIA	0	↑	DIPB	DIB	DIPA	DOA	?	?
WRITE_MODE_B=WRITE_FIRST											
0	↑	DIPA	DIA	1	↑	DIPB	DIB	?	?	DIPB	DIB
WRITE_MODE_A=WRITE_FIRST, WRITE_MODE_B=WRITE_FIRST											
0	↑	DIPA	DIA	1	↑	DIPB	DIB	?	?	?	?

Notes:

1. ADDRA=ADDRB, ENA=1, ENB=1, ?=Unknown or invalid data

Conflict Resolution

There is no dedicated monitor to arbitrate the result of identical addresses on both ports. The application must time the two clocks appropriately. However, conflicting simultaneous writes to the same location never cause any physical damage.

Block RAM Design Entry

Various tools help create Spartan-3 block RAM designs, two of which are the Xilinx CORE Generator system and VHDL or Verilog instantiation of the appropriate Xilinx library primitives.

Xilinx CORE Generator System

The Xilinx CORE Generator system provides both a Single Port Block Memory and a Dual Port Block Memory module generator, as shown in Figure 5. Both module generators support RAM, ROM, and Write Only functions, according to the control signals that are selected. Any size memory that can be created in the architecture is supported.

Both modules are parameterizable as with most CORE Generator modules. To create a module, specify the component name and choose to include or exclude control inputs, and choose the active polarity for the control inputs. For the Dual-Port Block Memory, once the organization or aspect ratio for Port A is selected, only the valid options for Port B are displayed.

Optionally, specify the initial memory contents. Unless otherwise specified, each memory location initializes to zero. Enter user-specified initial values via a Memory Initialization File, consisting of one line of binary data for every memory location. A default file is generated by the CORE Generator system. Alternatively, create a coefficients file (.coe), which not only defines the initial contents in a radix of 2, 10, or 16, but also defines all the other control parameters for the CORE Generator system.

The output from the CORE Generator system includes a report on the options selected and the device resources required. If a very deep memory is generated, some external multiplexing may be required, and these resources are reported as the number of logic slices required. In addition, the software reports the number of bits available in block RAM that are less than 100% utilized. For simulation purposes, the CORE Generator system creates VHDL or Verilog behavioral models.

- **CORE Generator**: **Single-Port Block Memory** module (RAM or ROM)
- **CORE Generator**: **Dual-Port Block Memory** module (RAM or ROM)

VHDL and Verilog Instantiation

VHDL and Verilog synthesis-based designs can either infer or directly instantiate block RAM, depending on the specific logic synthesis tool used to create the design.

Inferring Block RAM

Some VHDL and Verilog logic synthesis tools, such as the Xilinx Synthesis Tool (XST) and Synplicity Synplify both infer block RAM based on the hardware described. The Xilinx ISE Project Navigator includes templates for inferring block RAM in your design. To use the templates within Project Navigator, select **Edit → Language Templates** from the menu, and then select **VHDL** or **Verilog**, followed by **Synthesis Templates → RAM** from the selection tree. Finally, select the preferred block RAM template.

It is still possible to directly instantiate block RAM, even if portions of the design infer block RAM.

Instantiation Templates

For VHDL- and Verilog-based designs, various instantiation templates are available to speed development. Within the Xilinx ISE Project Navigator, select **Edit → Language Templates** from the menu, and then select **VHDL** or **Verilog**, followed by **Component Instantiation → Block RAM** from the selection tree.

The appendices include example code showing how to instantiate block RAM in both VHDL and Verilog.

In VHDL, each template has a component declaration section and an architecture section. Each part of the template must be inserted within the VHDL design file. The port map of the architecture section must include the signal names used in the application.

The SelectRAM_Ax templates (with x = 1, 2, 4, 9, 18, or 36) are single-port modules and instantiate the corresponding RAMB16_Sx module.

SelectRAM_Ax_By templates (with x = 1, 2, 4, 9, 18, or 36 and y = 1, 2, 4, 9, 18, or 36) are dual-port modules and instantiate the corresponding RAMB16_Sx_Sy module.

Initialization in VHDL or Verilog Codes

Block RAM memory structures can be initialized in VHDL or Verilog code for both synthesis and simulation. For synthesis, the attributes are attached to the block RAM instantiation and are copied within the EDIF output file compiled by Xilinx Alliance Series™ tools. The VHDL code simulation uses a **`generic`** parameter to pass the attributes. The Verilog code simulation uses a **`defparam`** parameter to pass the attributes.

The VHDL and Verilog examples in the appendices illustrate these techniques.

Block RAM Applications

Typically, block RAM is used for a variety of local storage applications. However, the following section describes additional, perhaps less obvious block RAM capabilities, illustrating some powerful capabilities to spur the imagination.

Creating Larger RAM Structures

Block SelectRAM columns have specialized routing to allow cascading blocks with minimal routing delays. Wider or deeper RAM structures incur a small delay penalty.

Block RAM as Read-Only Memory (ROM)

By tying the write enable input Low, block RAM optionally functions as registered block ROM. The ROM outputs are synchronous and require a clock input and perform exactly like a block RAM read operation. The ROM contents are defined by the initial contents at design time.

After design compilation, the ROM contents can also be updated using the Data2BRAM utility described below.

FIFOs

First-In, First-Out (FIFO) memories, also known as elastic stores, are perhaps the most common application of block RAM, other than for random data storage. FIFOs typically resynchronize data, either between two different clock domains, or between two parts of a system that have different data rates, even though they operate from a single clock. The Xilinx CORE Generator system provides two parameterizable FIFO modules, one a synchronous FIFO where both the read and write clocks are synchronous to one another and the other an asynchronous FIFO where the read and write clocks are different.

Application note XAPP261 demonstrates that the FIFO read and write ports can be different data widths, integrating the data width converter into the FIFO.

Application note XAPP291 describes a self-addressing FIFO that is useful for throttling data in a continuous data stream.

- **CORE Generator**: **Synchronous FIFO** module
- **CORE Generator**: **Asynchronous FIFO** module
- **XAPP258**: *FIFOs Using Block RAM*, includes reference design
- **XAPP261**: *Data-Width Conversion FIFOs Using Block RAM Memory*, includes reference design
- **XAPP291**: *Self-Addressing FIFO*

Storage for Embedded Processors

Block RAM also enables efficient embedded processor applications. RAM performs a variety functions in an embedded processor such as those listed below.

- Register file for processor register set, although for some processors, distributed RAM may be a preferred solution.
- Stack or LIFO for stack-based architectures and for call stacks.
- Fast, local code storage. The fast access time to internal block RAM significantly boosts the performance of embedded processors. However, on-chip storage is limited by the number of available block RAMs.
- Large dual-ported mailbox memory shared with external processor or DSP device.
- Temporary trace buffers (see **Circular Buffers, Shift Registers, and Delay Lines**) to ease and enhance application debugging.

Updating Block RAM/ROM Content by Directly Modifying Device Bitstream

In a typical design flow, the initial contents of block RAM/ROM is defined at design time and compiled into the device bitstream that is downloaded to and configures a Spartan-3 FPGA.

However, for some applications, the actual memory contents may not be known when the bitstream is created or may change later. One example is if a processor embedded with the Spartan-3 FPGA uses block RAM to store program code. To avoid re-compiling the FPGA design just to incorporate a code change, Xilinx provides a utility called Data2BRAM that updates an existing FPGA bitstream with new block RAM/ROM contents.

As shown in Figure 20, the inputs to Data2BRAM include the new RAM contents—typically the output from the embedded processor compiler/linker, the present FPGA bitstream, and a file that describes both the mapping between the system address space and the addressing used on the individual block RAMs and the physical location of each block RAM.

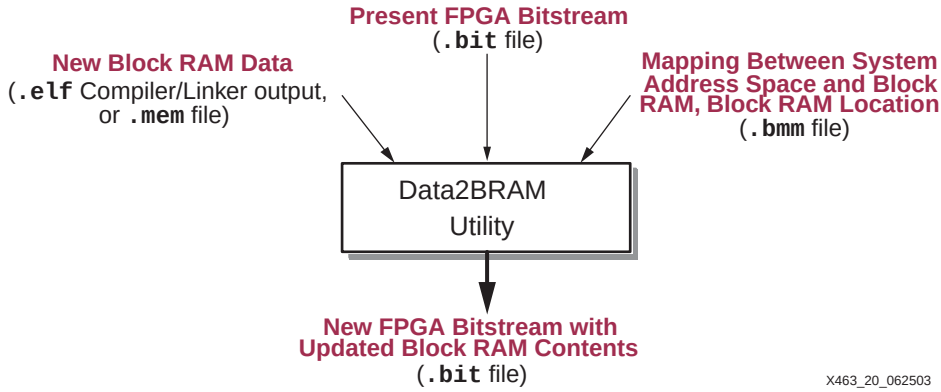


Figure 20: **The Data2BRAM Utility Updates Block RAM Contents in a Bitstream**

Two Independent Single-port RAMs Using One Block RAM

Some applications may require more single-port RAMs than there are RAM blocks on the device. However, a simple trick allows a single block RAM to behave as if it were two, completely independent single-port memories, effectively doubling the number of RAM blocks on the device. The penalty is that each RAM block is only half the size of the original block, up to 9K bits total.

Figure 21 shows how to create two independent single-port RAMs from one block RAM. Tie the most-significant address bit of one port High and the most-significant address bit of the other port Low. Both ports evenly split the available RAM between them.

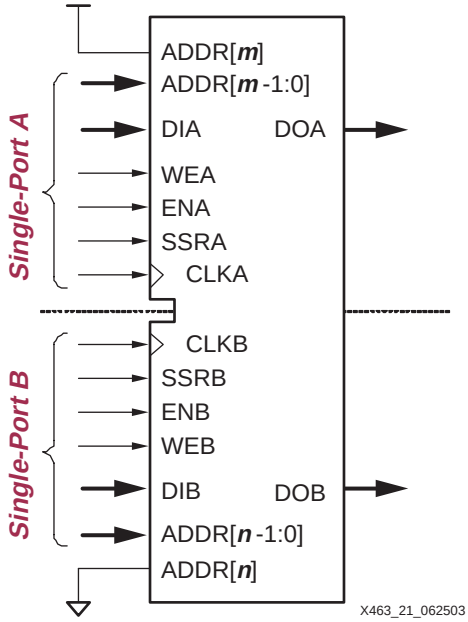


Figure 21: **One Block RAM Becomes Two Independent Single-Port RAMs**

Both ports are independent, each with its own memory organization, data inputs and outputs, clock input and control signals. For example, Port A could be 256x36 while Port B is 2Kx4.

Figure 21 splits the available memory evenly between the two ports. With additional logic on the upper address lines, the memory can be split into other ratios.

A 256x72 Single-Port RAM Using One Block RAM

Figure 22 illustrates how to create a 256-deep by 72-bit wide single-port RAM using a single block RAM. As in the previous example, the memory array is split into halves. One half contains the lower 36 bits and the upper half stores the upper 36 bits, effectively creating a 72-bit wide memory.

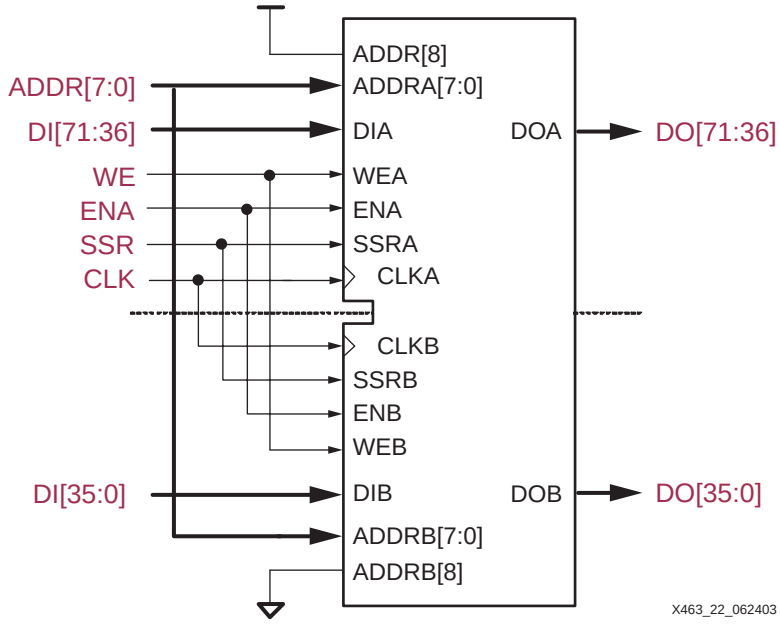


Figure 22: **A 256x72 Single-Port RAM Using a Single Block RAM**

The most-significant address line, ADDR[8] is tied High on one port and Low on the other. Both ports share the same the address inputs, control inputs, and clock input.

Circular Buffers, Shift Registers, and Delay Lines

Circular buffers are used in a variety of digital signal processing applications, such as finite impulse response (FIR) filters, multi-channel filtering, plus correlation and cross-correlation functions. Circular buffers are also useful simply for delaying data to resynchronize it with other parts of a data path.

Figure 23 conceptually describes how a circular buffer operates. Data is written into the buffer. After ***n*** clock cycles, that same data is clocked out of the buffer while new data is written to the same location.

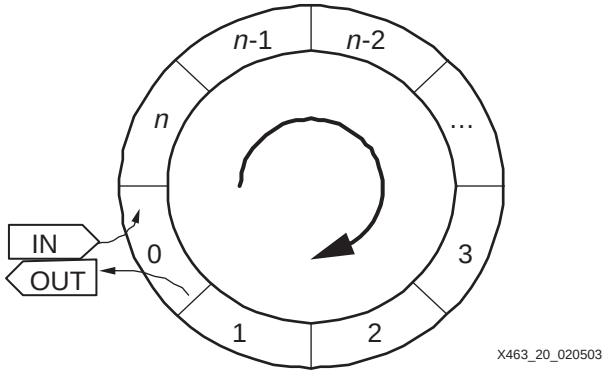


Figure 23: **Circular Buffer**

Figure 24 describes the hardware implementation to create a circular buffer using block RAM. A modulo-***n*** counter drives the address inputs to a single-port block RAM. For simple data delay lines, the block RAM writes new data on every clock cycle.

The circular buffer also reads the delayed data value on every clock edge. Using block RAM's READ_FIRST write mode, both the incoming write data and the outgoing read data use the same clock input and the same clock edge, both simplifying the design and improving overall performance. The actual write and read behavior is described in Figure 17.

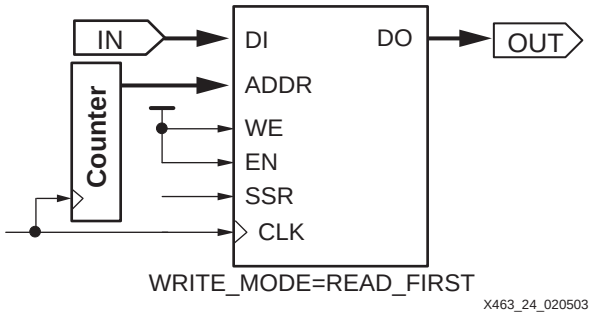


Figure 24: **Circular Buffer Implementation Using Block RAM and Counter**

In Figure 24, the width of the IN and OUT data ports is identical, although they do not need be. Using dual-port mode, the ports can be different widths. Figure 25 shows an example where byte-wide data enters the block RAM and a 32-bit word exits the block RAM. Furthermore, the data can be delayed up to 2,048 byte-clock cycles.

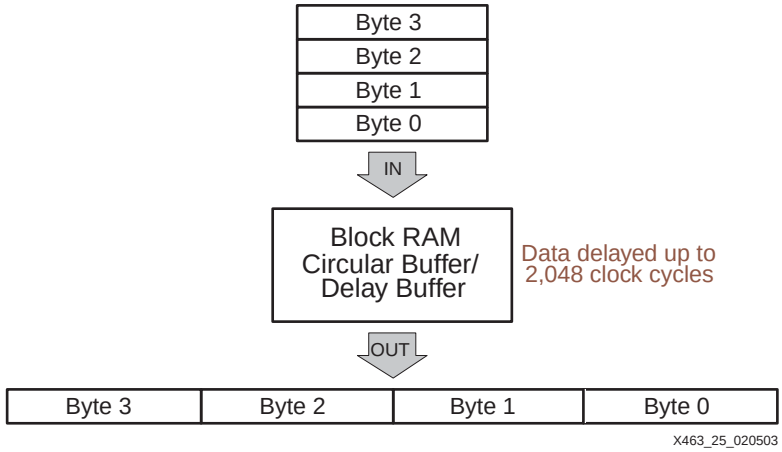


Figure 25: **Merge Circular Buffer and Port-Width Converter into a Single Block RAM**

A single block RAM is configured as dual-port memory. The incoming byte-wide data feeds Port B, which is configured as a 2Kx9 memory. The outgoing 32-bit data appears on Port A and consequently, Port A is configured as a 512x36 memory.

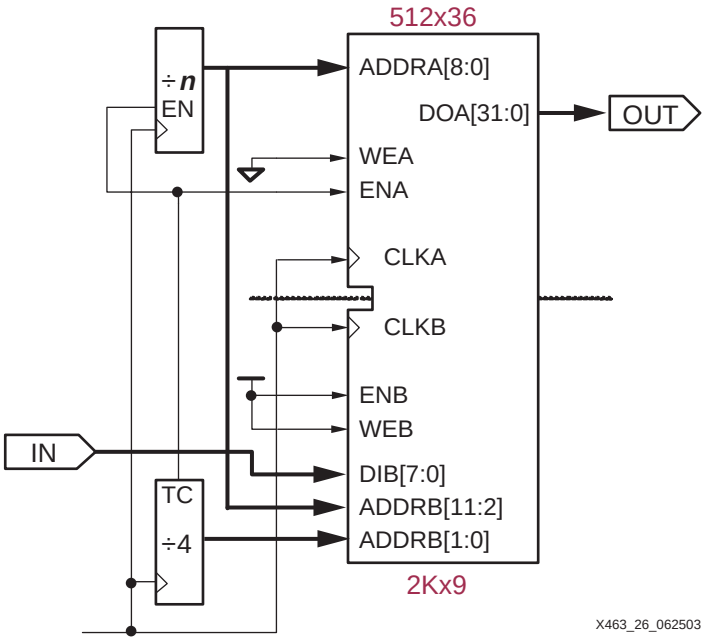


Figure 26: **Incoming Byte-Wide Data is Delayed 4n Clock Cycles, Converted to 32-Bit Data**

Manipulating the addresses that feeds both ports creates the 4***n***-byte clock delay. Every 32-bit output word requires four incoming bytes. Consequently, a divide-by-4 counter feeds the two lower address bits, ADDRB[1:0]. After four bytes are stored, a terminal count, TC, from the lower counter enables Port A plus a separate divide-by-***n*** counter. The enable signal latches the 32-bit output data on Port B and increments the upper counter. The combination of the divide-by-4 counter and the divide-by-***n*** counter effectively create a divide-by-4***n*** counter. The output from the divide-by-***n*** counter forms the more-significant address bits to Port B, ADDRB[11:2] and the entire address to Port A, ADDRA[9:0].

Fast Complex State Machines and Microsequencers

Because block RAMs can be configured with any set of initial values, they also make excellent dual-ported registered ROMs that can be used as state machines. For example, a 128-state, 8-way branch finite state machine with 38 total state outputs, fits in a single block RAM, as shown in Figure 27.

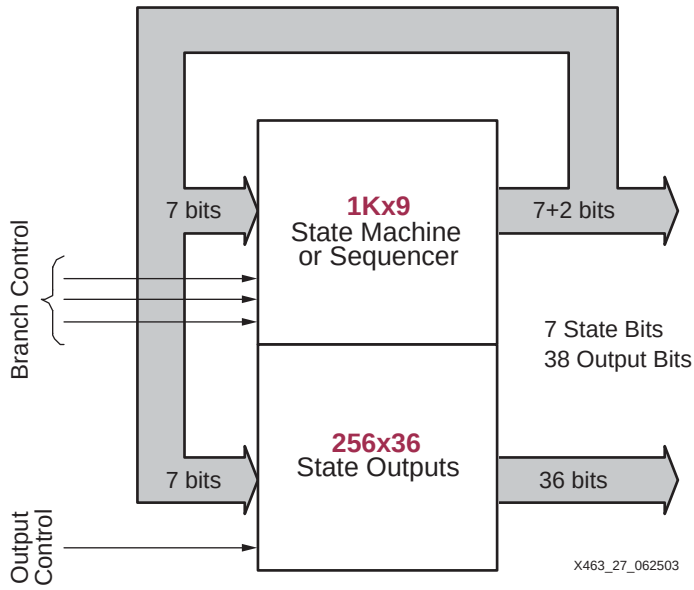


Figure 27: **128-State Finite State Machine with 38 Outputs in a Single Block RAM**

A dual-port block RAM memory is divided into two completely independent half-size, single-port memories by tying the most-significant address bit of one port High and the other one Low, similar to Figure 21. Port A is configured as 2Kx9 but used as a 1K x 9 single-port ROM. Seven outputs feed back as address inputs, stepping through the 128 states. The 1Kx9 ROM has ten total address lines, seven of which are the current-state inputs and the remaining three address inputs determine the eight-way branch. Any of the 128 states can conditionally branch to any set of eight new states, under the control of these three address inputs.

Port B is configured as 512 x 36 and used as a 256 x 36 single-port ROM. It receives the same 7-bit current-state value from Port A, and drives 36 outputs that can be arbitrarily defined for each state. However, due to the synchronous nature of block ROM, the 36 outputs from the 256x36 ROM are delayed by one clock cycle. The eighth address input can invoke an alternate definition of the 36 outputs. Two additional state bits are available from the 1Kx9 block, but are not delayed by one clock.

This same basic architecture can be modified to form a 256-state finite state machine with four-way branch, or a 64-state state machine with 16-way branch.

If additional branch-control inputs are needed, they can be combined using an input multiplexer. The advantages of this design are its low cost (a single block RAM), its high performance (125+ MHz), the absence of lay-out or routing issues, and complete design freedom.

Fast, Long Counters Using RAM

A counter is an example of a simple state machine, where the next state depends only on the current state. A binary up counter, for example, simply increments the current state to create the next state. Figure 28 shows a 20-bit binary up counter, with clock enable and synchronous reset, implemented in a single block RAM.

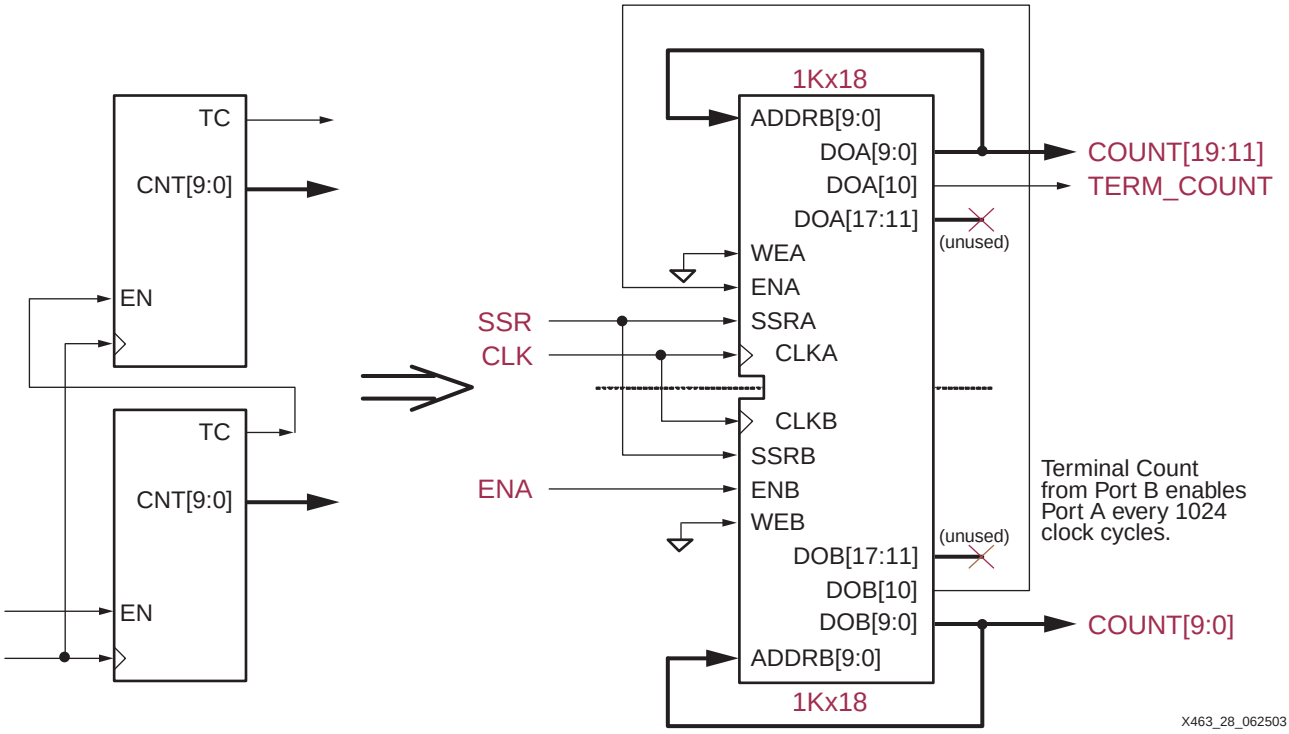


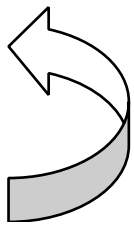
Figure 28: **Two 10-Bit Counters Create a 20-Bit Binary Counter Using a Single Block RAM**

A 20-bit binary counter can be constructed from two identical 10-bit binary counters, with the lower 10-bit counter enabling the upper 10-bit counter every 1024 clock cycles. In this example, Port B is a 1Kx18 ROM (WEB is Low) that forms the lower 10-bit counter. The ten less-significant data outputs, representing the current state, connect directly to the ten address inputs, ADDRB[9:0]. The next state is looked up in the ROM using the current state applied to the address pins. The eleventh data bit, D[10], forms the terminal-count output from the counter. In this example, the upper seven data bits, DOB[17:11] are unused.

The next-state logic for a binary counter appears in Table 12. The counter starts at state 0—or the value specified by the INIT or SRVAL attributes—and counts through to 0x3FF (1023 decimal) at which time the terminal count, D[10], is active and the counter rolls over back to 0.

Table 12: **Next-State Logic for Binary Up Counter**

Current State	State Outputs	Next State
	TC	COUNT
ADDR[9:0] (Hex)	D[10]	D[9:0] (Hex)
0	0	1
1	0	2
2	0	3
…	…	…
3FFF	1	0

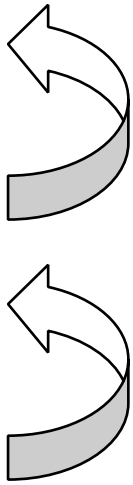


Port A is configured nearly identically to Port B, except that Port A is enabled by the terminal count output from Port B. The 10-bit counter in Port A has the identical counting pattern as Port B, except that it increments at 1/1024th the rate of Port B.

With a simple modification, the 20-bit up counter becomes an 18-bit up/down counter. Using the most-significant address input as a direction control, the same basic counter architecture either increments or decrements its count, as shown in Table 13. In this example, the counter increments when the Up/Down control is Low and decrements when High. The ROM memory is split between the incrementing and decrementing next-state logic.

Table 13: **Next-State Logic for Binary Up/Down Counter**

Up/Down Control	Present State	State Outputs	Next State
		TC	COUNT
ADDR[9]	ADDR[8:0] (Hex)	D[10]	D[9:0] (Hex)
0 (Up)	0	0	1
	1	0	2
	2	0	3
	…	…	…
	1FFF	1	0
1 (Down)	1FFF	0	1FFE
	1FFE	0	1FFD
	1FFD	0	1FFC
	…	…	…
	0	1	1FFF



Various other counter implementations are possible including the following.

- Binary up and up/down counters of various modulos determined by the combinations of the modulos of the counters implemented in Port A and Port B.
- Counters with other incrementing and decrementing patterns including fast gray-code counters.
- A six-digit BCD counter in one block ROM, configured as 512x36, plus one CLB.

Four-Port Memory

Each block RAM is physically a dual-port memory. However, due to the block RAM's fast access performance, it is possible to create multi-port memories by time-division multiplexing the signals in and out of the memory. A block RAM with some additional logic easily supports up to four ports but at the cost of additional access latency for each port. The following application note provides additional details and a reference design.

- **XAPP228**: *Quad-Port Memories in Virtex Devices*, includes reference design

Content-Addressable Memory (CAM)

Content-Addressable Memory (CAM), sometimes known as associative memory, is used in a variety of networking and data processing applications. In most memory applications, content is referenced by an address. In CAM applications, the content is the driving input and the output indicates whether or not the content exists in memory and, if so, provides a reference to its location.

An easy way to envision how a CAM operates is to think of an index to a book. Looking up an item, *i.e.*, the content, first determines whether the item exists in the index and if it does, provides a reference to its location, *i.e.*, the page number of where the item can be found.

- **CORE Generator**: Content-Addressable Memory module
- XAPP260: *Using Block RAM for High-Performance Read/Write CAMs*
- XAPP201: *An Overview of Multiple CAM Designs*, written for Virtex/E and Spartan-II/E architectures but provides a useful overview to the techniques involved

Implementing Logic Functions Using Block RAM

Inside every Spartan-3 logic cell, there is a four-input RAM/ROM called a look-up table or LUT. The LUT performs any possible logic function of its four inputs and forms the basis of the Spartan-3 logic architecture.

Another possible application for block RAM is as a much larger look-up table. In one of its organizations, a block RAM—used as ROM in this case—has 14 inputs and a single output. Consequently, block RAM is capable of implementing any possible arbitrary logic function of up to 14 inputs, regardless of the complexity and regardless of inversions. There are a few restrictions, however.

- There cannot be any asynchronous feedback paths in the logic, such as those that create latches.
- The logic output must be synchronized to a clock input. Block RAM does not support asynchronous read outputs.

If the logic function meets these requirements, then a single block RAM implements the following functions.

- Any possible Boolean logic function of up to 14 inputs
- Nine separate arbitrary Boolean logic functions of 11 inputs, as long as the inputs are shared.
- Various other combinations are possible, but may have restrictions to the number of inputs, the number of shared inputs, or the complexity of the logic function.

Due to the flexibility and speed of CLB logic, block RAM may not be faster or more efficient for simple wide functions like an address decoder, where multiple inputs are ANDed together. Block RAM will be faster and more efficient for complex logic functions, such as majority decoders, pattern matching, correlators.

Fuzzy Pattern Matching Circuit Example

For example, Figure 29 illustrates a fuzzy pattern matching circuit that detects both exact matches and those patterns that are close enough. Each incoming bit is matched against the required MATCH pattern. Then, any “don’t care” bits are masked off, indicating that the specific bit should always match. Then, the number of matching bits is counted and compared against an activation threshold. If the number of matching bits is greater than the activation threshold, then the input data mostly matches the required pattern and the MATCH output goes High.

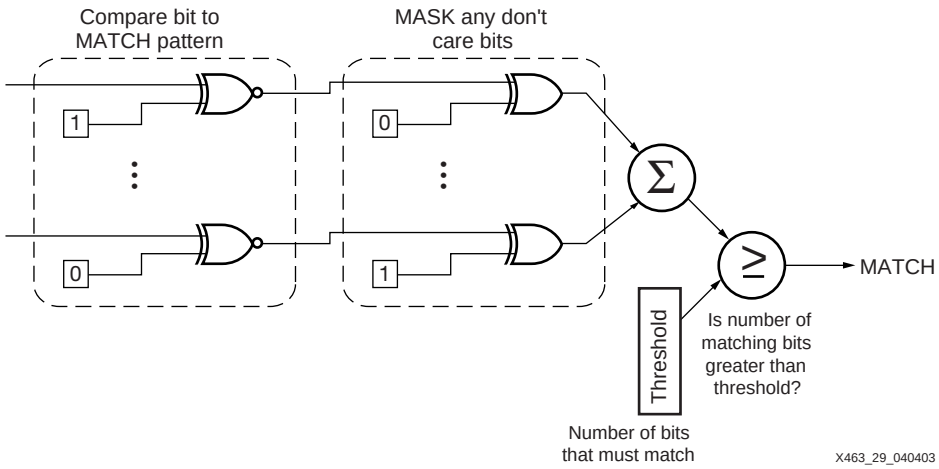


Figure 29: **A 14-Input Fuzzy Pattern Matching Circuit Implemented in a Single Block RAM**

If the application requires a new matching pattern or different logic function, it could be loaded via the second memory port.

Implemented in CLB logic, this function would require numerous logic cells and multiple layers of logic. However, because the MATCH, MASK, and Threshold values are known in advance, the function can be pre-computed and then stored in block RAM. For each input condition, *i.e.*, starting at address 0 and incremented through the entire memory, the output condition can be pre-computed. A 14-input fuzzy pattern matching circuit requires a single block RAM and performs the operation in a single clock cycle.

Mapping Logic into Block RAM Using `MAP` –bp Option

The Xilinx ISE software does not automatically attempt to map logic functions into block RAM. However, there is a mapping option to aid the process.

The block RAM mapping option is enabled when using the MAP –bp option. If so enabled, the Xilinx ISE logic mapping software attempts to place LUTs and attached flip-flops into an unused single-output, single-port block RAM. The final flip-flop output is required as block RAMs have a synchronous, registered output. The mapping software packs the flip-flop with whatever LUT logic is driving it. No register will be packed into block RAM without LUT logic, and *vice versa*.

To specify which register outputs will be converted to block RAM outputs, create a file containing a list of the net names connected to the register output(s). Set the environment variable `XIL_MAP_BRAM_FILE` to the file name, which instructs the mapping software to use this file. The MAP program looks for this environment variable whenever the –bp option is specified. Only those output nets listed in the file are converted into block RAM outputs.

- **PCs**:
  `set XIL_MAP_BRAM_FILE=file_name`
- **Workstations**:
  `setenv XIL_MAP_BRAM_FILE file_name`

Waveform Storage, Function Tables, Direct Digital Synthesis (DDS) Using Block RAM

Another powerful block RAM application is waveform storage, including function tables such as trigonometric functions like sine and cosine. Sine and cosine form the backbone of other functions such as direct digital synthesis (DDS) to generate output waveforms. The Xilinx CORE Generator system provides parameterizable modules for both:

- **CORE Generator**: **Sine/Cosine Look-Up Table** module
- **CORE Generator**: **Direct Digital Synthesizer (DDS)** module

Another potential application of waveform storage is in various signal companders (compressors/expanders) and normalization circuits used to boost important parts of a signal within the available bandwidth. Examples include converters between linear data, u-Law encoded data, and A-Law encoded data commonly used in telecommunications.

The dual-port nature of block RAM not only facilitates waveform storage, it also enables an application to update the waveform, either with a completely new waveform or with corrected or normalized waveform data. In the example shown in Figure 30, Port A initially contains the currently active waveform. The application can load a new waveform on Port B.

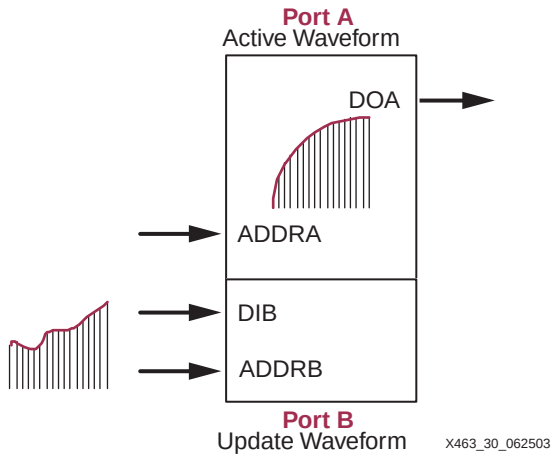


Figure 30: **Dual-Port Block RAM Facilitates Waveform Storage and Updates**

As in real-world engineering, sometimes it is faster to look up an answer than deriving it. The same is true in digital designs. Block RAM is also useful for storing pre-computed function tables where the output, y, is a function of the input, x, or y=f(x).

For example, instead of creating the CLB logic that implements the following polynomial equation, the function can be pre-computed and stored in a block RAM.

$$Y = Ax^3 - Bx^2 + Cx + D$$

The values A, B, C, and D are all constants. The output, y, depends only on the input, x. The output value can be pre-computed for each input value of x and stored in memory. There are obvious limitations as the function may not fit in a single logic block either because of the range of values for x, or the magnitude of the output, y. For example, a 512x36 block ROM implements the above equation for input values between 0 and 511. The range of x is limited by its exponential effect on y. With x at its maximum value for this specific example, y requires at least 28 output bits.

Some other look-up functions possible in a single block RAM/ROM include the following.

- Various complex arithmetic functions of a single input, including mixtures of functions such as log(x), square-root(x). Multipliers of two values are possible but are typically limited by the number of block RAM inputs. The Spartan-3 embedded 18x18 multipliers are a better solution for pure multiplication functions.
- Two independent 11-bit binary to 4-digit BCD converters, with the block ROM configured as 1Kx18. The least-significant bit (LSB) of each converter bypasses the ROM as the converted result is the same as the original value, *i.e.* the LSB indicates whether the value is odd or even.
- Two independent 3-digit BCD to 10-bit binary converters, with the block ROM configured as 2Kx9 and the LSBs bypass the converters.
- Sine-cosine look-up tables using one port for sine, the other one for cosine, with 90 degree-shifted addresses, 18-bit amplitude, 10-bit angular resolution.
- Two independent 10-bit binary to three-digit, seven-segment LED output converter with the block ROM configured as 1Kx18. Leading zeros are displayed as blanks. Because input values are limited to 1023, the LED digits display from “0” to “3FF”. Consequently, the logic for the most-significant digit requires only four inputs (segment a=d=g, segment f is always High).

Related Materials and References

- "**Using Leftover Multipliers and Block RAM in Your Design**" by Peter Alfke, Xilinx, Inc.
  http://www.xilinx.com/support/techxclusives/leftover-techX11.htm
- "**The Myriad Uses of Block RAM**" by Jan Gray, Gray Research, LLC.
  http://www.fpgacpu.org/usenet/bb.html
- **Libraries Guide**, for Xilinx ISE 5.2i by Xilinx, Inc.
  - *Adobe Acrobat [PDF]*
    http://toolbox.xilinx.com/docsan/xilinx5/pdf/docs/lib/lib.pdf, pp. 1593–1640.

    If ISE 5.2i is installed in the default directory, this document is also located in the following path or within Project Navigator by selecting **Help→Online Documentation**. When the Acrobat document appears, click **Libraries Guide** from the table of contents on the left.
    `C:\Xilinx\doc\usenglish\docs\lib\lib.pdf`
  - `RAMB16_Sn` *Primitive [HTML]*
    http://toolbox.xilinx.com/docsan/xilinx5/data/docs/lib/lib0371_355.html
  - `RAMB16_Sm_Sn` *Primitive [HTML]*
    http://toolbox.xilinx.com/docsan/xilinx5/data/docs/lib/lib0372_356.html

Conclusion

The Spartan-3 FPGA's abundant, fast, and flexible block RAMs provide invaluable on-chip local storage for scratchpad memories, FIFOs, buffers, look-up tables, and much more. Using unique capabilities, block RAM implements such functions as shift registers, delay lines, counters, and wide, complex logic functions.

Block RAM is supported in applications using the broad spectrum of Xilinx ISE development software, including the CORE Generator system and can be inferred or directly instantiated in VHDL or Verilog synthesis designs.

Appendix A: VHDL Instantiation Example

The following VHDL instantiation example is for the Synopsys FPGA Express system. The example XC3S_RAMB_1_PORT module uses the **SelectRAM_A36.vhd** VHDL template. This and other templates are available for download from the following Web link. The following example is a VHDL code snippet and will not compile as is.

- ftp://ftp.xilinx.com/pub/applications/xapp/xapp463_vhdl.zip

```
-- Module: XC3S_RAMB_1_PORT
-- Description: 18Kb Block SelectRAM example
-- Single Port 512 x 36 bits
-- Use template "SelectRAM_A36.vhd"
--
-- Device: Spartan-3 Family
--------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
--
-- Syntax for Synopsys FPGA Express
-- pragma translate_off
library UNISIM;
use UNISIM.VCOMPONENTS.ALL;
-- pragma translate_on
--
entity XC3S_RAMB_1_PORT is
port (
    DATA_IN     : in std_logic_vector (35 downto 0);
    ADDRESS     : in std_logic_vector (8 downto 0);
    ENABLE      : in std_logic;
    WRITE_EN    : in std_logic;
    SET_RESET   : in std_logic;
    CLK         : in std_logic;
    DATA_OUT    : out std_logic_vector (35 downto 0)
```

```
);
end XC3S_RAMB_1_PORT;
--
architecture XC3S_RAMB_1_PORT_arch of XC3S_RAMB_1_PORT is
--
-- Components Declarations:
--
component BUFG
port (
    I : in std_logic;
    O : out std_logic
);
end component;
--
-- Syntax for Synopsys FPGA Express
component RAMB16_S36
-- pragma translate_off
generic (
-- "Read during Write" attribute for functional simulation
WRITE_MODE : string := "READ_FIRST" ; -- WRITE_FIRST(default)/ READ_FIRST/
NO_CHANGE
-- Output value after configuration
INIT : bit_vector(35 downto 0) := X"000000000";
-- Output value if SSR active
SRVAL : bit_vector(35 downto 0) := X"012345678";
-- Initialize parity memory content
INITP_00 : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000FEDCBA9876543210";
INITP_01 : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000000000000000000";
... (snip)
INITP_07 : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000000000000000000";
-- Initialize data memory content
INIT_00 : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000FEDCBA9876543210";
INIT_01 : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000000000000000000";
... (snip)
INIT_3F : bit_vector(255 downto 0) :=
X"0000000000000000000000000000000000000000000000000000000000000000"
);
-- pragma translate_on
port (
    DI    : in std_logic_vector (31 downto 0);
    DIP   : in std_logic_vector (3 downto 0);
    ADDR  : in std_logic_vector (8 downto 0);
    EN    : in STD_LOGIC;
    WE    : in STD_LOGIC;
    SSR   : in STD_LOGIC;
    CLK   : in STD_LOGIC;
    DO    : out std_logic_vector (31 downto 0);
    DOP   : out std_logic_vector (3 downto 0)
);
end component;
--
-- Attribute Declarations:
attribute WRITE_MODE : string;
attribute INIT: string;
attribute SRVAL: string;
-- Parity memory initialization attributes
attribute INITP_00: string;
attribute INITP_01: string;
... (snip)
attribute INITP_07: string;
-- Data memory initialization attributes
attribute INIT_00: string;
attribute INIT_01: string;
```

```
... (snip)
attribute INIT_3F: string;
--
-- Attribute "Read during Write mode" = WRITE_FIRST(default)/ READ_FIRST/
NO_CHANGE
attribute WRITE_MODE of U_RAMB16_S36: label is "READ_FIRST";
attribute INIT of U_RAMB16_S36: label is "000000000";
attribute SRVAL of U_RAMB16_S36: label is "012345678";
--
-- RAMB16 memory initialization for Alliance
-- Default value is "0" / Partial initialization strings are padded
-- with zeros to the left
attribute INITP_00 of U_RAMB16_S36: label is
"0000000000000000000000000000000000000000000000000FEDCBA9876543210";
attribute INITP_01 of U_RAMB16_S36: label is
"0000000000000000000000000000000000000000000000000000000000000000";
... (snip)
attribute INITP_07 of U_RAMB16_S36: label is
"0000000000000000000000000000000000000000000000000000000000000000";
--
attribute INIT_00 of U_RAMB16_S36: label is
"0000000000000000000000000000000000000000000000000FEDCBA9876543210";
attribute INIT_01 of U_RAMB16_S36: label is
"0000000000000000000000000000000000000000000000000000000000000000";
... (snip)
attribute INIT_3F of U_RAMB16_S36: label is
"0000000000000000000000000000000000000000000000000000000000000000";
--
-- Signal Declarations:
--
-- signal VCC : std_logic;
-- signal GND : std_logic;
signal CLK_BUFG: std_logic;
signal INV_SET_RESET : std_logic;
--
begin
-- VCC <= '1';
-- GND <= '0';
--
-- Instantiate the clock Buffer
U_BUFG: BUFG
port map (
    I => CLK,
    O => CLK_BUFG
);
--
-- Use of the free inverter on SSR pin
INV_SET_RESET <= NOT SET_RESET;
-- Block SelectRAM Instantiation
U_RAMB16_S36: RAMB16_S36
port map (
    DI   =>  DATA_IN (31 downto 0),   -- insert 32 bits data-in bus (<31 downto 0>)
    DIP  =>  DATA_IN (35 downto 32),  -- insert 4 bits parity data-in bus (or <35
                                      --    downto 32>)
    ADDR =>  ADDRESS (8 downto 0),    -- insert 9 bits address bus
    EN   =>  ENABLE,                  -- insert enable signal
    WE   =>  WRITE_EN,                -- insert write enable signal
    SSR  =>  INV_SET_RESET,           -- insert set/reset signal
    CLK  =>  CLK_BUFG,                -- insert clock signal
    DO   =>  DATA_OUT (31 downto 0),  -- insert 32 bits data-out bus (<31 downto 0>)
    DOP  =>  DATA_OUT (35 downto 32)  -- insert 4 bits parity data-out bus (or <35
                                      --    downto 32>)
);
--
end XC3S_RAMB_1_PORT_arch;
```

Appendix B: Verilog Instantiation Example

The following Verilog instantiation example is for the Synopsys FPGA Express system. The example XC3S_RAMB_1_PORT module uses the **SelectRAM_A36.v** Verilog template. This and other templates are available for download from the following Web link. The following example is a Verilog code snippet and will not compile as is.

- ftp://ftp.xilinx.com/pub/applications/xapp/xapp463_verilog.zip

```
// Module: XC3S_RAMB_1_PORT
// Description: 18Kb Block SelectRAM-II example
// Single Port 512 x 36 bits
// Use template "SelectRAM_A36.v"
//
// Device: Spartan-3 Family
//----------------------------------------------------------------------
module XC3S_RAMB_1_PORT (CLK, SET_RESET, ENABLE, WRITE_EN, ADDRESS, DATA_IN,
DATA_OUT);
    input CLK, SET_RESET, ENABLE, WRITE_EN;
    input [35:0] DATA_IN;
    input [8:0] ADDRESS;
    output [35:0] DATA_OUT;
    wire CLK_BUFG, INV_SET_RESET;
//Use of the free inverter on SSR pin
assign INV_SET_RESET = ~SET_RESET;
// initialize block ram for simulation
// synopsys translate_off
defparam
//"Read during Write" attribute for functional simulation
U_RAMB16_S36.WRITE_MODE = "READ_FIRST", //WRITE_FIRST(default)/ READ_FIRST/ NO_CHANGE
//Output value after configuration
U_RAMB16_S36.INIT = 36'h000000000,
//Output value if SSR active
U_RAMB16_S36.SRVAL = 36'h012345678,
//Initialize parity memory content
U_RAMB16_S36.INITP_00 =
256'h0123456789ABCDEF0000000000000000000000000000000000000000000000000,
U_RAMB16_S36.INITP_01 =
256'h0000000000000000000000000000000000000000000000000000000000000000,
... (snip)
U_RAMB16_S36.INITP_07 =
256'h0000000000000000000000000000000000000000000000000000000000000000,
//Initialize data memory content
U_RAMB16_S36.INIT_00 =
256'h0123456789ABCDEF0000000000000000000000000000000000000000000000000,
U_RAMB16_S36.INIT_01 =
256'h0000000000000000000000000000000000000000000000000000000000000000,
... (snip)
U_RAMB16_S36.INIT_3F =
256'h0000000000000000000000000000000000000000000000000000000000000000;
// synopsys translate_on
//Instantiate the clock Buffer
BUFG U_BUFG ( .I(CLK), .O(CLK_BUFG));
//Block SelectRAM Instantiation
RAMB16_S36 U_RAMB16_S36 (
    .DI(DATA_IN[31:0]),
    .DIP(DATA_IN-PARITY[35:32]),
    .ADDR(ADDRESS),
    .EN(ENABLE),
    .WE(WRITE_EN),
    .SSR(INV_SET_RESET),
    .CLK(CLK_BUFG),
    .DO(DATA_OUT[31:0]),
    .DOP(DATA_OUT-PARITY[35:32]));
// synthesis attribute declarations
/* synopsys attribute
WRITE_MODE "READ_FIRST"
INIT "000000000"
SRVAL "012345678"
```

```
INITP_00
"0123456789ABCDEF0000000000000000000000000000000000000000000000000000"
INITP_01
"0000000000000000000000000000000000000000000000000000000000000000000000"
... (snip)
INITP_07
"0000000000000000000000000000000000000000000000000000000000000000000000"
INIT_00
"0123456789ABCDEF0000000000000000000000000000000000000000000000000000"
INIT_01
"0000000000000000000000000000000000000000000000000000000000000000000000"
... (snip)
INIT_3F
"0000000000000000000000000000000000000000000000000000000000000000000000"
*/
endmodule
```

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
04/05/03	1.0	Initial Xilinx release.
05/12/03	1.1	Updated block RAM information for the XC3S50 device. Updated Table 1, Figure 2, Figure 18, Figure 26, and the text associated with the **Introduction** and **Block RAM Location and Surrounding Neighborhood** sections. Added 256x72 single-port mode to Table 2. Updated hypertext links in **Related Materials and References**, **Appendix A: VHDL Instantiation Example**, and **Appendix B: Verilog Instantiation Example**. Corrected Figure 22. DIB and ADDRB connections were swapped.
06/25/03	1.1.1	Minor editing changes.
07/23/03	1.1.2	Improved display quality for Figure 5.