

Xilinx Answer 46945

Data2Mem Usage and Debugging Guide

Important Note: This downloadable PDF of an Answer Record is provided to enhance its usability and readability. It is important to note that Answer Records are Web-based content that are frequently updated as new information becomes available. Please visit the Xilinx Technical Support Website and review [\(Xilinx Answer 46945\)](#) for the latest version of this solution.

Introduction

Data2MEM is a Xilinx® tool that allows initialization of block RAM (BRAM) without the need of reimplementing designs. So, if a change in software is required after the design has been implemented, the use of Data2MEM provides a tool to re-initialize BRAM without a need to re-implement the whole design. Data2MEM uses various input files and generates updated output files (including BIT and HDL files).

Data2MEM uses a command line tool for quick generation of updated BIT files. The command line tool can also optionally generate formatted text files of BIT and ELF files. Data2MEM generates HDL files (Verilog and VHDL) for pre and post synthesis simulation. More information on this command line tool can be found in [1].

This document steps through examples to illustrate how Data2MEM is used. It describes how to write a BMM file, how to dump a BIT file and an ELF file, and then verify the contents of the block RAM in the design. Data2MEM could be used both as standalone and with EDK. This document describes the usage of Data2MEM in both cases. The main objectives of this document are to provide users with a conceptual background on how and where Data2MEM is used, and also provide a comprehensive description for users to be able to debug Data2MEM issues on their own.

Data2Mem Input and Output Files

This section discusses the various input and output files of Data2MEM. The files discussed are as follows:

- Block RAM Memory Map Files (.bmm)
- Executable and Linkable Format Files (.elf)
- Memory Files (.mem)
- Bitstream Files (.bit)
- Hardware Descriptive Language Files, Verilog (.v) and VHDL (.vhd)
- User Constraints File (.ucf)

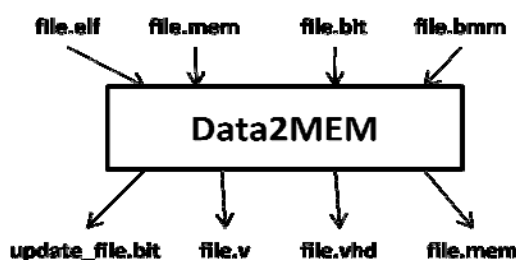


Figure 1: Data2MEM Input and Output Files

Block RAM Memory Map Files (.bmm)

The BMM file is a text file that describes how individual block RAMs make up a contiguous logical space, and is designed in text syntax form for user readability and editing. It describes the memory to be used to map the executable ELF File. The BMM description includes; memory size, data width, addresses range, and the BRAM locations on the physical FPGA.

BMM uses block structures by keywords or directives to maintain structures in groups, or blocks of data, that describe address space, bus groupings, and comments. Keywords are case-sensitive and are uppercase. Improved readability is achieved by symbolic name usage and key words to refer to groups or entities.

BMM files can be created in multiple ways; these include manually using Data2MEM to generate BMM file templates and automated scripting. Manually creating a BMM file will be detailed in the BMM File section below. A BMM file can be edited directly as it is a text file and supports both `//` and `/*...*/` commenting styles. The `/*...*/` comment style comments out a block of text and can be nested. The `//` comment style comments out everything to the end of the current line.

Numbers can be entered either in decimal or hexadecimal notation. The `0x#####` notation is used when using hexadecimal numbers.

The example BMM syntax below shows the text based syntax used to describe the organization of a simple RAM16 with address space `0x00000000 - 0x000007FF`, 16K deep by 32 bits wide.

```
ADDRESS_SPACE v4_minor RAMB16 [0x00000000:0x000007FF]

BUS_BLOCK

v4_block_memory/BU2/U0/blk_mem_generator/valid.cstr/ramloop[0].ram.r/v4_noinit.ram/SP.W
IDE_PRIM.SP [31:0] PLACED = X7Y13;

END_BUS_BLOCK;
END_ADDRESS_SPACE;
```

Executable and Linkable Format Files (.elf)

ELF files are generated by software compiler and linker tools (e.g., Xilinx® Software Development Kit (SDK)). ELF files are binary data files that contain an application image. Since they contain binary data they cannot be directly edited. Data2MEM uses ELF files as its basic data input form.

The following command displays the content of an ELF file:

```
data2mem -bd test.elf -d
```

These dumps are used primarily for debugging purposes.

Memory Files (.mem)

MEM files are text files that describe continuous blocks of data. They can be edited directly like BMM files and also support both `//` and `/*...*/` commenting styles. Data2MEM uses MEM files for both data input and output.

MEM files use an industry standard format that consists of two basic elements. The two elements are Hexadecimal Address Specifier and Hexadecimal Data Values. The Hexadecimal Address Specifier is indicated by the “@” symbol followed by a Hexadecimal Address Value (e.g., `@0FFF`).

If the number of digits in a Hexadecimal Address Value is an odd number, the value of the first Hexadecimal value is assumed to be a zero (e.g., `@48F`) will become `@048F`.

The address of each successive data value depends on whether the MEM file is being used as an input or an output. A detailed description of the differences between the MEM file being used as an input or an output is described in the following *Data2MEM User Guide*: http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/data2mem.pdf

([Xilinx Answer 14384](#)) provides more information on how to write a MEM file.

The MEM file can also be set up to use parity. A description of this is also available in the *Data2MEM User Guide*. It is important to note that when using parity, the values in memory might not be the same as the expected values. This is due to Data2MEM assuming the upper (most significant) Bit Lane data bits are connected to the parity data bits of a block RAM. This is described in more detail in the Parity Bits section below.

MEM File as Input

MEM files used as inputs have the following format restrictions:

- All the values in contiguous blocks of data are treated as continuous streams of bits, so everything between the adjacent data values is ignored.
- An address specifier that resides within an address space range is defined by the BMM file.
- If an address gap exists between two contiguous blocks of data, the address gap is assumed to be nonexistent memory.
- No two contiguous blocks of data can overlap an address range.
- A contiguous block of data must fit within a single address space range defined in a BMM file.

MEM File as Output

Output MEM files are primarily used for Verilog simulations with third-party memory models.

- All data values must be the same number of bits wide and must be the same width as expected by the memory model.
- Data values reside within a larger array of values, starting at zero. An address specifier is an index offset from the beginning of a larger array of where the data should be and is not a true address.
- If an address gap exists between two contiguous blocks of data, the data between the gaps still logically exists, but is undefined.

An `OUTPUT` keyword can be inserted for outputting memory device MEM files as shown below:

```
top/ram_cntlr/ram0 [7:0] OUTPUT = ram0.mem;
```

BIT Files (.bit)

BIT files contain the bit image that can be downloaded to the FPGA device. BIT files are binary and cannot be edited directly, and are generated initially by Xilinx[®] implementation tools. A text dump of the BIT file can also be generated by using the appropriate command line tool prompt.

The following command displays the text dump of a BIT file:

```
data2mem -bm test.bmm -bt test.bit -d
```

These dumps are used primarily for debugging purposes.

NOTE: Data2MEM can only update BIT files that have been generated without encryption or compression options.

Hardware Descriptive Language Files

Data2MEM can output a text file as Verilog (.v) and VHDL (.vhd). The Verilog file contains the `defparam` records to initialize block RAMs. The VHDL file contains the `bit_vector` constraints to initialize the block RAMs. Both Verilog and VHDL are primarily used for pre and post synthesis simulation. Editing of these files is not recommended.

User Constraint File (.UCF)

Data2MEM also outputs a UCF file that contains the INIT constraints to initialize the block RAMs.

Initializing Block RAM in ISE Design Suite

This section discusses generation and instantiation of block RAMs in an ISE® tools project. The example shown below instantiates 2 x Single Port ROMs. The two Single Port ROMs that are generated by the CORE Generator™ software are bram0 16(Read Width) x 4096(Read Depth) and bram1 32(Read Width) x 1024(Read Depth) Single Port ROMs. The VHDL code below shows the instantiated ROMs:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

- - Uncomment the following library declaration if using
- - arithmetic functions with Signed or Unsigned values
- - use IEEE.NUMERIC_STD.ALL;

- - Uncomment the following library declaration if instantiating
- - any Xilinx primitives in this code.
- - library UNISIM;
- - use UNISIM.VComponents.all;

entity top is
PORT (
    clka : IN STD_LOGIC;
    addra : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
    douta : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    clk b : IN STD_LOGIC;
    addrb : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
    doutb : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
);
end top;

architecture Behavioral of top is

COMPONENT bram0
PORT (
    clka : IN STD_LOGIC;
    addra : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
    douta : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
);
END COMPONENT;

COMPONENT bram1
PORT (
    clka : IN STD_LOGIC;
    addra : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
    douta : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
);
END COMPONENT;

begin

bram0_instance : bram0
PORT MAP (
    clka => clka,
    addra => addra,
    douta => douta
);

bram1_instance : bram1
PORT MAP (
    clka => clk b,
    addra => addrb,
    douta => doutb
);

end Behavioral;
```

BRAM0 - 16x4096 Single Port ROM Generation in the CORE Generator software

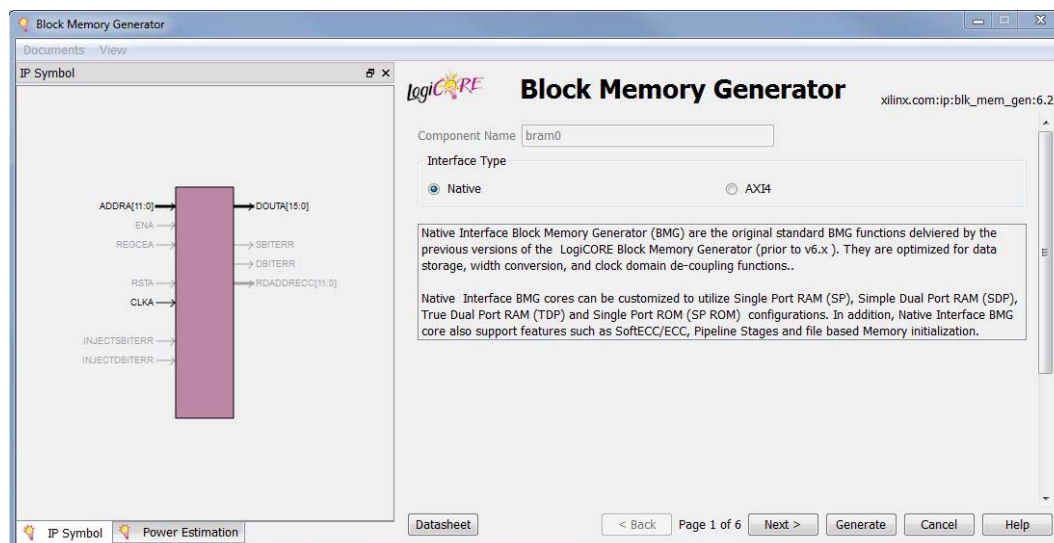


Figure 2: Block Memory Generator for 16 x4096 Single Port ROM, Page 1 of 6

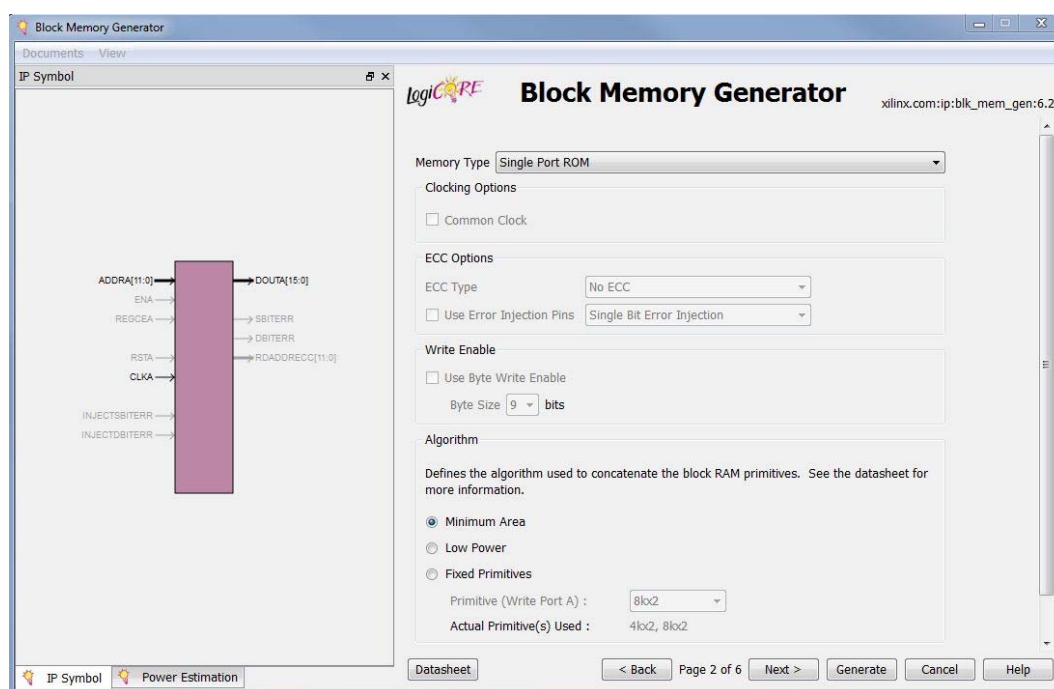


Figure 3: Block Memory Generator for 16 x4096 Single Port ROM, Page 2 of 6

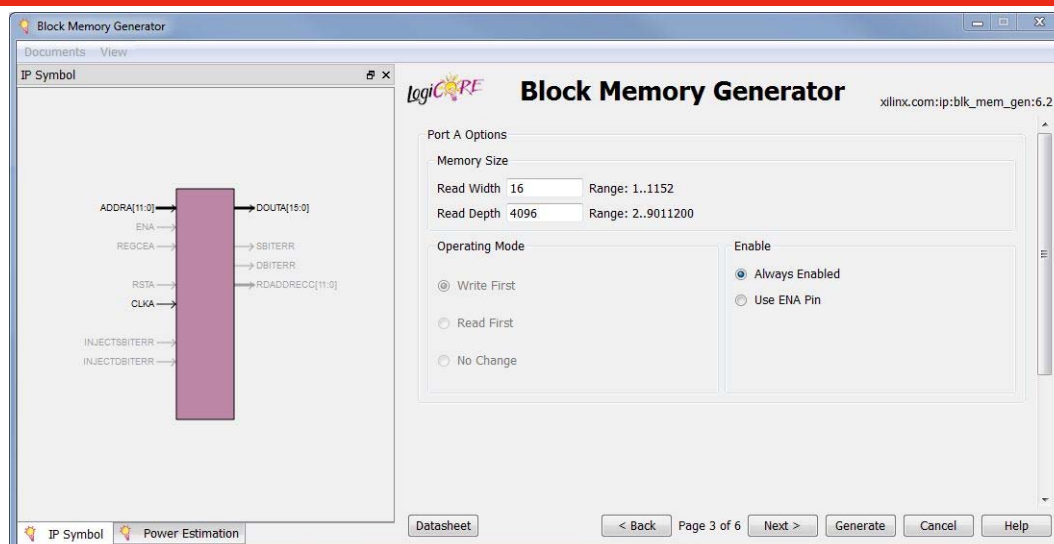


Figure 4: Block Memory Generator for 16 x4096 Single Port ROM, Page 3 of 6

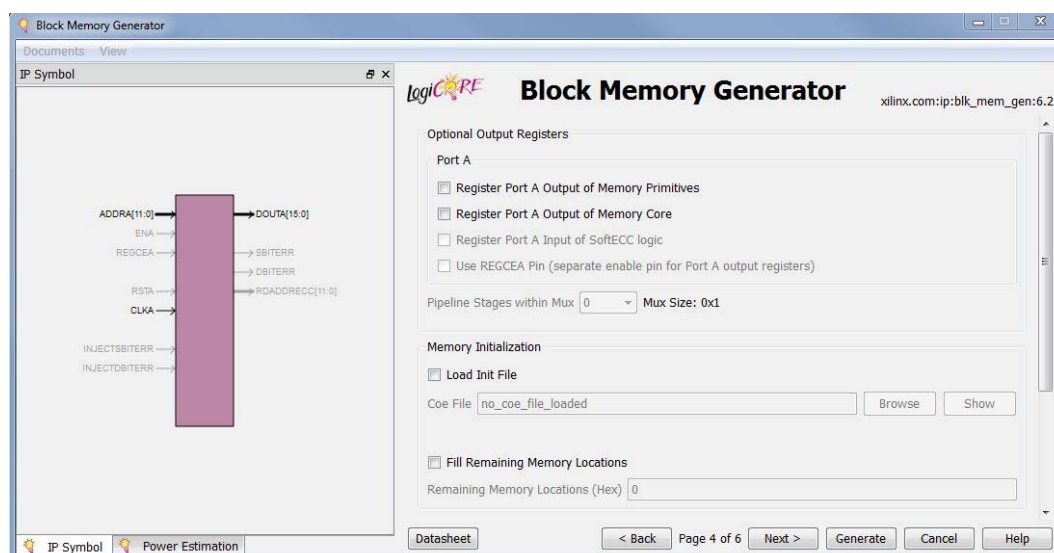


Figure 5: Block Memory Generator for 16 x4096 Single Port ROM, Page 4 of 6

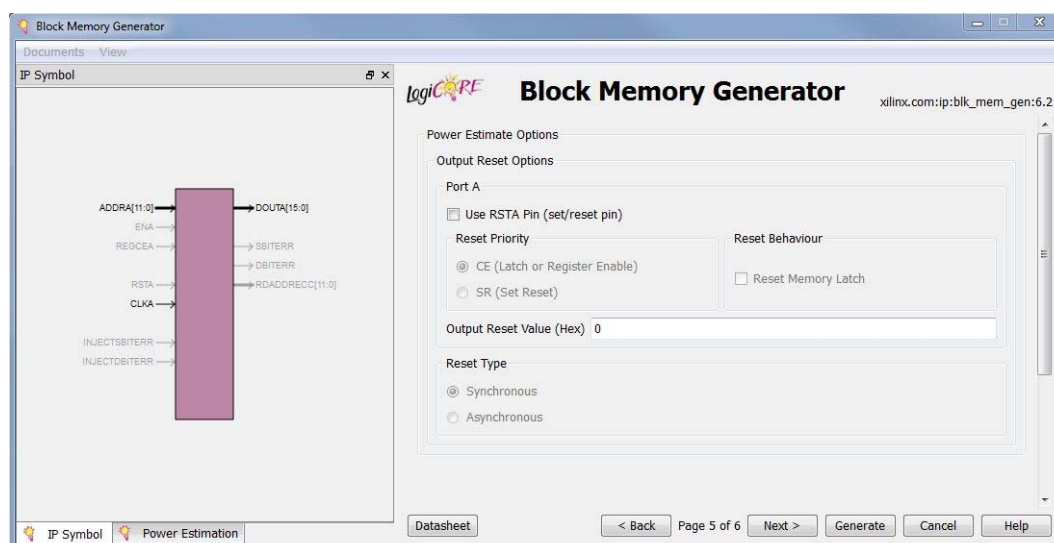


Figure 6: Block Memory Generator for 16 x4096 Single Port ROM, Page 5 of 6

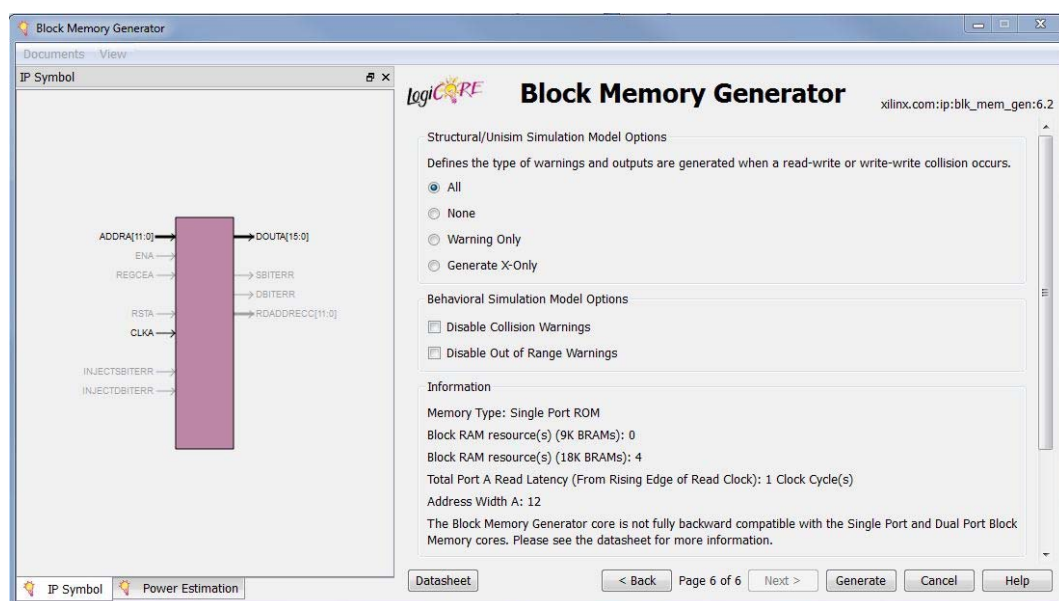


Figure 7: Block Memory Generator for 16 x 4096 Single Port ROM, Page 6 of 6

BRAM1 - 32x1024 Single Port ROM Generation in the CORE Generator software

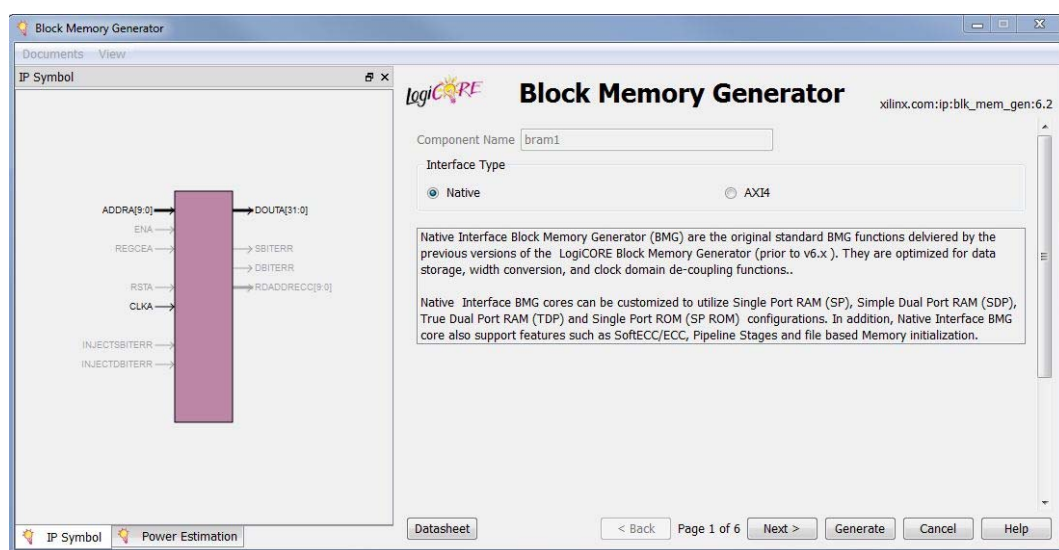


Figure 8: Block Memory Generator for 32 x 1024 Single Port ROM, Page 1 of 6

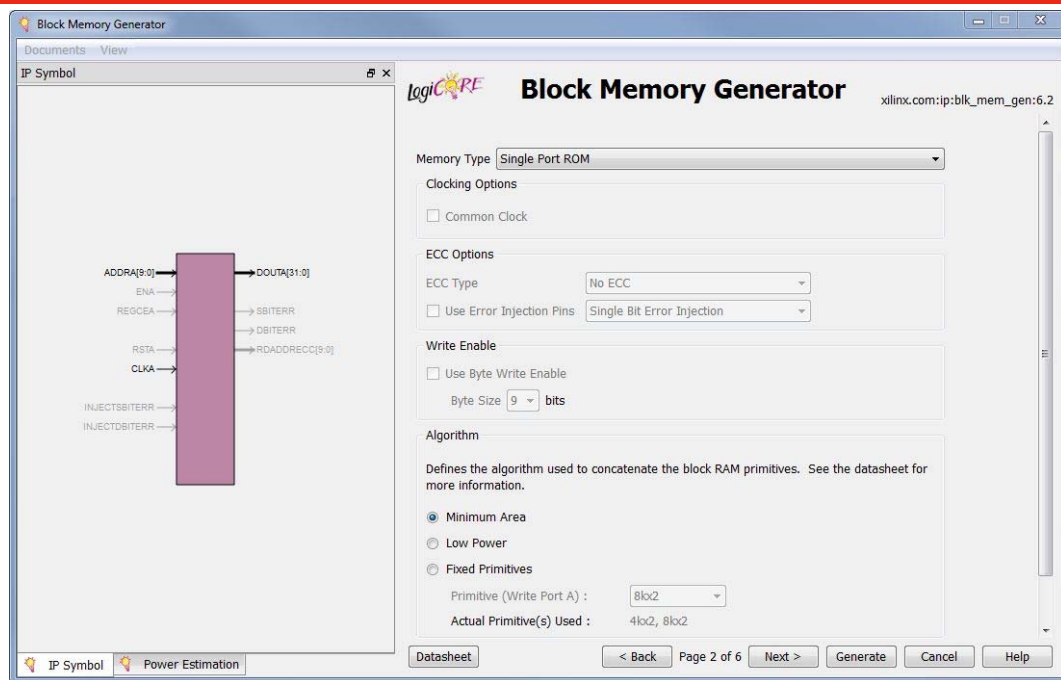


Figure 9: Block Memory Generator for 32 x1024 Single Port ROM, Page 2 of 6

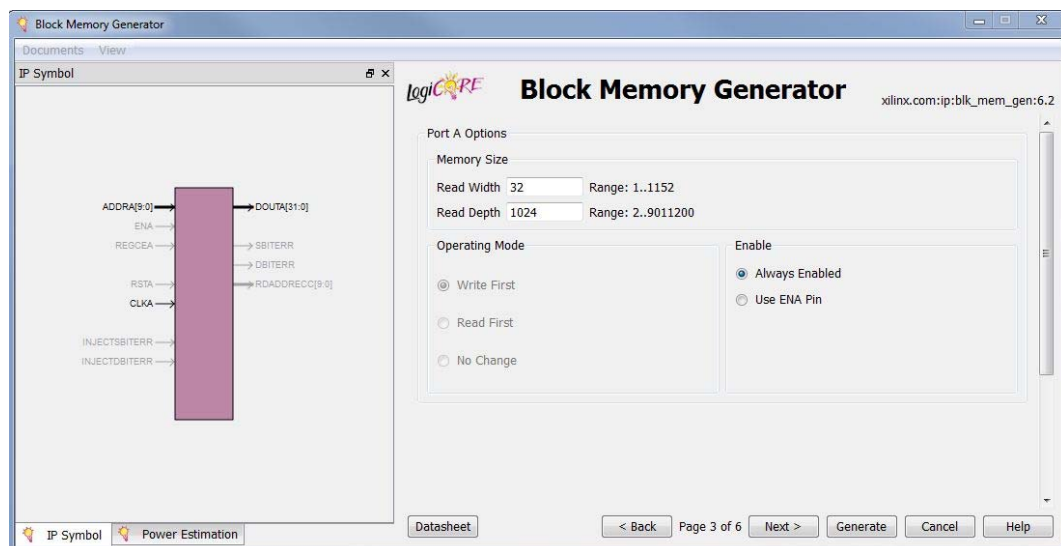


Figure 10: Block Memory Generator for 32 x1024 Single Port ROM, Page 3 of 6

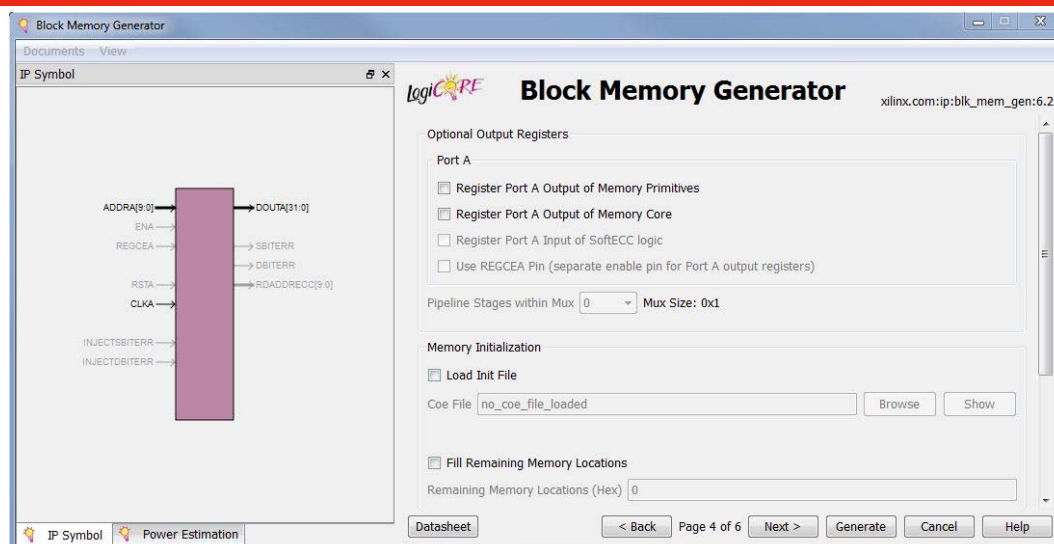


Figure 11: Block Memory Generator for 32 x1024 Single Port ROM, Page 4 of 6

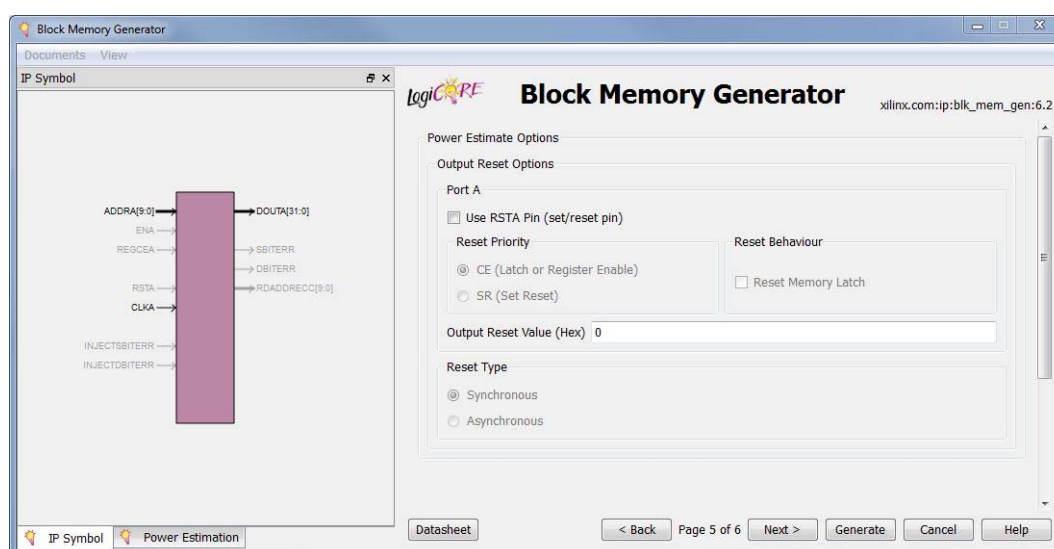


Figure 12: Block Memory Generator for 32 x1024 Single Port ROM, Page 5 of 6

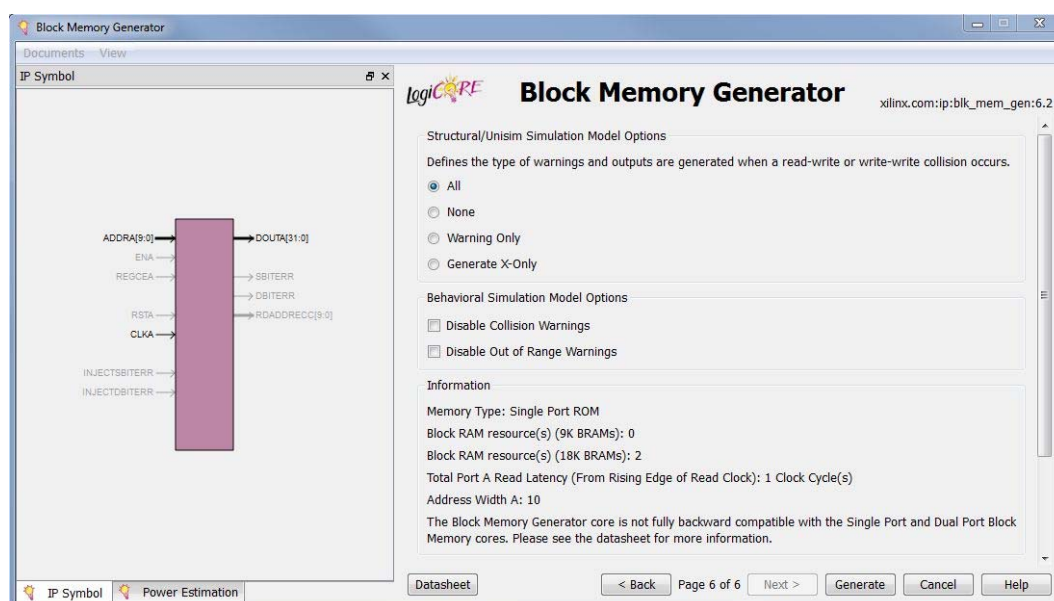


Figure 13: Block Memory Generator for 32 x1024 Single Port ROM, Page 6 of 6

BMM File

The different sections of the BMM file and the appropriate syntax for each section is described in the following sections. A more detailed description of these sections is provided in [1].

Address Map Definitions (Multiple Processor Support)

Multiple Processors are supported in Data2MEM by using the following keywords:

- ADDRESS_MAP
- END_ADDRESS_SPACE.

These Address Map keywords surround the address space and have the following syntax:

```
ADDRESS_MAP map_name processor_type processor_ID
    ADDRESS_SPACE space_name mtype[start:end]
    .
    .
    END_ADDRESS_MAP;
.
END_ADDRESS_MAP;
```

Each processor has its own ADDRESS_MAP definition. ADDRESS_SPACE names must be unique only within a single ADDRESS_MAP. Data2MEM requires instance names to be unique within an entire BMM file.

- map_name is an identifier that refers to all the ADDRESS_SPACE keywords in an ADDRESS_MAP.
- processor_type specifies the processor type.
- iMPACT uses processor_ID as a JTAG ID to download external memory contents to the proper processor.

Address tags take on two forms and can be substituted wherever an ADDRESS_SPACE name was previously used.

- ADDRESS_SPACE is referred to by its map_name and space_name separated by a period or a dot character. For example: cpu1.bram
- The address tag name can be shortened to just the ADDRESS_MAP name, which confines the data translation to only those ADDRESS_SPACES within the named ADDRESS_MAP. So, this is used to send data to a specific processor without having to name each individual ADDRESS_SPACE.

For backwards compatibility, ADDRESS_SPACE can still be defined outside of an ADDRESS_MAP structure. Those ADDRESS_SPACE keywords are assumed to belong to an unnamed ADDRESS_MAP definition of type MB, PPC405, or PPC440 and processor ID 0. Address tags for these ADDRESS_SPACES are used as the space_name.

If no ADDRESS_MAP tag names are supplied, data translation takes place for each ADDRESS_SPACE in all ADDRESS_MAP keywords with matching address ranges.

Address Space Definitions

A single contiguous address space is defined within the ADDRESS_SPACE and END_ADDRESS_SPACE keywords block. The name that follows the ADDRESS_SPACE keyword provides a symbolic name for the entire address space and refers to the entire contents of the address space. Multiple ADDRESS_SPACE definitions even for the same address space may be defined in the BMM file as long as each ADDRESS_SPACE name is unique. The Address Space definitions that belong to a memory map for a single processor are surrounded by the Address Space keywords and have the following syntax:

```
ADDRESS_SPACE ram_cntlr RAMB16 <WORD_ADDRESSING> [start_addr:end_addr]
.
.
END_ADDRESS_SPACE;
```

To utilize parity in block RAMs the <WORD_ADDRESSING> keyword must be used. This keyword tells Data2MEM that the smallest addressable unit is the bit lane width.

ADDRESS_SPACE Memory Device Types

The type of memory device used in the ADDRESS_SPACE is defined by a keyword. Listed below are the memory device types:

- RAM16
- RAM19
- RAM32
- RAM36
- MEMORY
- COMBINED

The correct keyword must be used for the memory size and style selected.

- RAM16 defines the memory as a 16-Kbit block RAM without parity.
- RAM18 defines the memory as a 18-Kbit block RAM using parity.
- RAM32 defines the memory as a 32-Kbit block RAM without parity.
- RAM36 defines the memory as a 36-Kbit block RAM using parity.
- MEMORY defines the memory device as generic memory. The size of the memory device is derived from the address range defined by the ADDRESS_SPACE.
- COMBINED - A BMM address space is defined for every memory controller. Non-contiguous addresses are addressed by using the keyword COMBINED. Data2MEM is instructed by the BMM address space syntax to combine physical address ranges into a single logical address space. The following example shows a 12k address space for two 32-bit bus memory controllers. One memory controller has two block RAMs and is configured with 16-bit data buses, the second memory controller has four block RAMs and is configured with 8-bit data buses. The following is the format of a COMBINED ADDRESS_SPACE:

```
ADDRESS_SPACE bram_block COMBINED [0x00000000:0x00002FFF]
  ADDRESS_RANGE RAMB16
    BUS_BLOCK
      bram_elab1/bram0 [31:16];
      bram_elab1/bram1 [15:0];
    END_BUS_BLOCK;
  END_ADDRESS_RANGE;
  ADDRESS_RANGE RAMB16
    BUS_BLOCK
      bram_elab2/bram0 [31:24];
      bram_elab2/bram1 [23:16];
      bram_elab2/bram2 [15:8];
      bram_elab2/bram3 [7:0];
    END_BUS_BLOCK;
  END_ADDRESS_RANGE;
END_ADDRESS_SPACE;
```

Bus Block (Bus Accesses) and Bit Lane

Bus Blocks are a variable number of sub-block definitions that are defined within the ADDRESS_SPACE. Block RAM Bit Lane definitions are defined within the Bus Blocks. The lowest addressable Bus Block is defined first and the highest addressable Bus Block is defined last. The Bus blocks have the following syntax:

```
BUS_BLOCK
  Bit_lane_definition
  Bit_lane_definition
END_BUS_BLOCK;
```

The particular block RAMs are assigned to bits in a CPU bus access using the Bit Lane definitions. The block RAM instance name is followed by the bit numbers the Bit Lane occupies. Bit Lane definitions have the following syntax:

```
BRAM_instance_name [MSB_bit_num:LSB_bit_num];
```

The physical row and column location within the FPGA device can also be indicated by getting the instance name (with hierarchy) and location from FPGA Editor. This is described in more detail later. To place the block RAM to a specific location in the FPGA device, the keyword `LOC` is used. If a back-annotated BMM file is created by the Xilinx® implementation tools, the `PLACED` keyword is inserted. The following is an example of Bit line syntax with FPGA locations identified:

```
top/ram_cntlr/ram0 [7:0] LOC = X3Y5; (Using LOC)
top/ram_cntlr/ram0 [7:0] PLACED = X3Y5; (Using PLACED)
```

Parity Bits

When using parity, Data2MEM assumes the parity bits occupy the upper (most significant) bits of the device data bus. Data2MEM discards the first two bits and uses the second two bits for parity.

In the example below, of the data to be stored to memory from a MEM file, the first two bits (**RED**) of each line of data in the MEM file are discarded and the second two bits (**BLUE**) of each line of data are used as the parity bits. The rest of the data is stored to block RAM Data memory. By substituting the hexadecimal value of 2 in front of the desired data provides an easy to understand representation of what is happening.

After the first two bits are discarded from the data, the second two bits are stored in the first two bits of the block RAM Parity block of memory. Then, the following data bits are stored in the block RAM data block of memory.

MEM File:

```
@00000000
2DEAD      (= 0010 1101 1110 1010 1101) - RED - Discarded BLUE - Used for Parity
2BEEF      (= 0010 1011 1110 1110 1111) - RED - Discarded BLUE - Used for Parity
21234      (= 0010 0001 0010 0011 0100) - RED - Discarded BLUE - Used for Parity
25678      (= 0000 0101 0110 0111 1000) - RED - Discarded BLUE - Used for Parity
```

BMM File:

```
ADDRESS_SPACE bram_block COMBINED [0x00000000:0x000035FF]
  ADDRESS_RANGE RAMB36
    BUS_BLOCK
      ../ramloop[0].ram.r/SP.SIMPLE_PRIM36.ram [17:0] LOC = X0Y22;
    END_BUS_BLOCK;
  END_ADDRESS_RANGE;
  ADDRESS_RANGE RAMB36
    BUS_BLOCK
      ../ramloop[1].ram.r/SP.SIMPLE_PRIM36.ram [17:0] LOC = X0Y21;
    END_BUS_BLOCK;
  END_ADDRESS_RANGE;
  ADDRESS_RANGE RAMB36
    BUS_BLOCK
      ../ramloop[2].ram.r/SP.SIMPLE_PRIM36.ram [17:0] LOC = X0Y23;
    END_BUS_BLOCK;
  END_ADDRESS_RANGE;
END_ADDRESS_SPACE;
```

Therefore, the block RAM Parity block of memory at 0x00000000 is 1010 1000, or A8 in Hexadecimal. This leaves the following 16 bits of data to be stored in the block RAM data memory location starting at 0x00000000 (shown in Figure 14).

```

BRAM data, Column 00, Row 22. Design instance "PCU_ROM_1_inst/U0/xst_blk_mem_generator/gnativebmg.native_blk_m
00000000: DE AD BE EF 12 34 56 78 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
||
BRAM parity.
00000000: A8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

Figure 14: BRAM Instances Memory Dump Text File

BMM File Generation

BMM File for 23 x 1024 and 16 x 4096 Single Port ROMs

The first thing that needs to be assessed and worked out before BMM generation is the address space. To calculate the address space, it is required to get the Memory Size (Read Width and Read Depth) of each of the block memories that were generated by the Block Memory Generator in the CORE Generator software. These values are the ones that are set on Page 3 of 6 of the Block Memory Generator in the CORE Generator software. This is shown in Figure 15 and Figure 16.

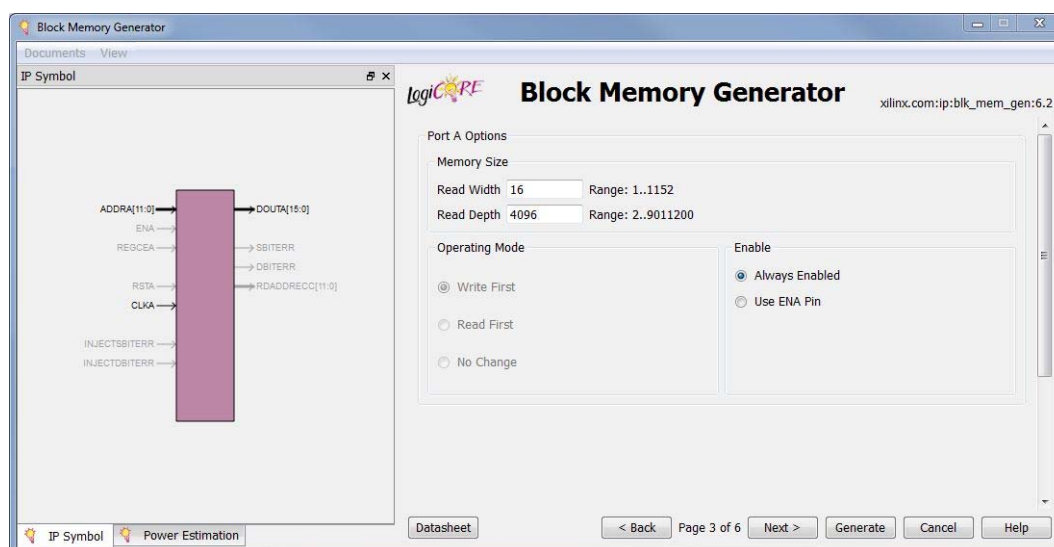


Figure 15: Block Memory Generator for 16 x4096 Single Port ROM, Page 3 of 6

For the memory block in Figure 15, the total memory that is addressed is calculated as follows:

16 (Read Width) * 4,096 (Read Depth) = 65,536 or 64Kbit.
 Now convert to bytes: 64Kbit / 8 = 8Kbytes or 2000 Hex

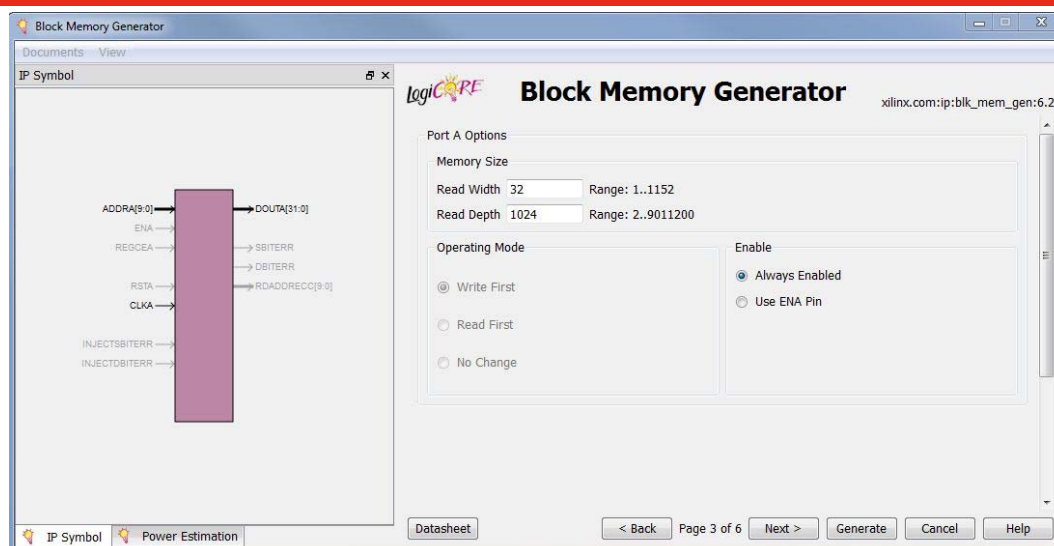


Figure 16: Block Memory Generator for 32 x1024 Single Port ROM, Page 3 of 6

For the memory block in Figure 16, the total memory that is addressed is calculated as follows:

$32 \text{ (Read Width)} * 1,024 \text{ (Read Depth)} = 32,768 \text{ or } 32\text{Kbit.}$
 Now convert to bytes: $32\text{Kbit} / 8 = 4\text{Kbytes or } 1000 \text{ Hex}$

With these calculations, it can be seen that the total address range of 3000 Hex is required to address both memory blocks with the bram0 being addressed from 0x00000000 to 0x00001FFF, and bram1 being addressed from 0x00002000 to 0x00002FFF.

The next thing is to find out the instance names of the generated block RAMs to create the BMM file. These can be retrieved in either the FPGA Editor or in PlanAhead™ in the ISE tools. The following sections detail getting the BRAM instance names in the FPGA Editor and PlanAhead tool.

FPGA Editor

To open FPGA Editor in the ISE software, select **"Tools" -> "FPGA Editor" -> "Post-Map or Place and Route"**. FPGA Editor will open with an "Array Window", "List Window" and "World Window" on view.

- 1) The "Array Window" displays a graphical representation of the FPGA device. The device components and the interconnections (both logical and routed) between these components are displayed in this window.
- 2) The "List Window" displays a list of components, nets, layers, paths, macros, constraints, and clock regions in a design. The pull-down drop box at the top of the window can be used to specify the items that are required to be displayed in the "List Window"
- 3) The "World Window" shows the area of the device that is currently displayed in the "Array Window".

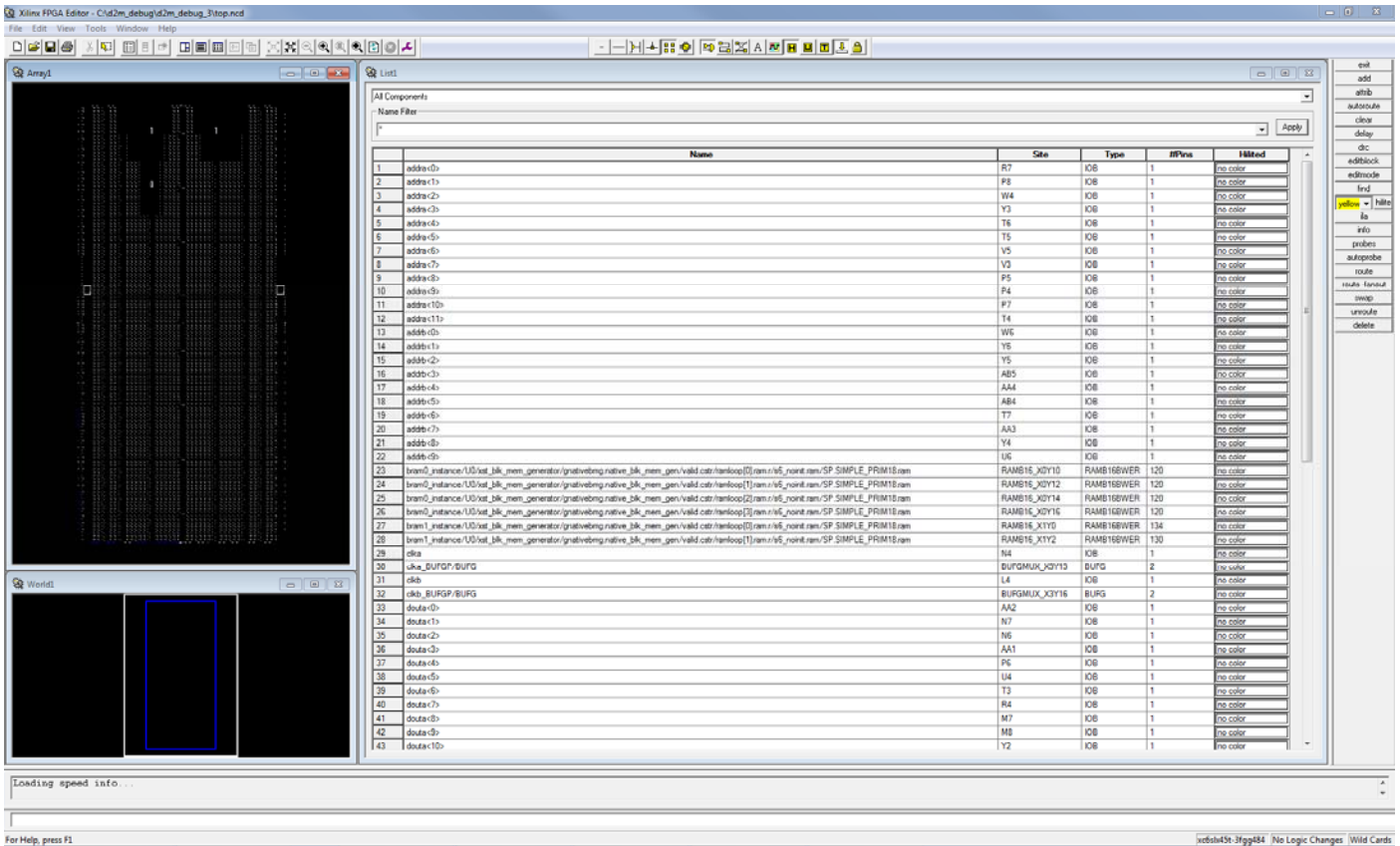



Figure 17: FPGA Editor

Figure 18 shows a screenshot of the required block RAM components in the “List Window.”

22	addb<9>	U6	IOB	1	no color
23	bram0_instance/U0/xst_blk_mem_generator/gnativebmg.native_blk_mem_gen/valid.ctr/ramloop[0].ram.r/s6_noinit.ram/SP.SIMPLE_PRIM18.ram	RAMB16_X0Y10	RAMB16BWER	120	no color
24	bram0_instance/U0/xst_blk_mem_generator/gnativebmg.native_blk_mem_gen/valid.ctr/ramloop[1].ram.r/s6_noinit.ram/SP.SIMPLE_PRIM18.ram	RAMB16_X0Y12	RAMB16BWER	120	no color
25	bram0_instance/U0/xst_blk_mem_generator/gnativebmg.native_blk_mem_gen/valid.ctr/ramloop[2].ram.r/s6_noinit.ram/SP.SIMPLE_PRIM18.ram	RAMB16_X0Y14	RAMB16BWER	120	no color
26	bram0_instance/U0/xst_blk_mem_generator/gnativebmg.native_blk_mem_gen/valid.ctr/ramloop[3].ram.r/s6_noinit.ram/SP.SIMPLE_PRIM18.ram	RAMB16_X0Y16	RAMB16BWER	120	no color
27	bram1_instance/U0/xst_blk_mem_generator/gnativebmg.native_blk_mem_gen/valid.ctr/ramloop[0].ram.r/s6_noinit.ram/SP.SIMPLE_PRIM18.ram	RAMB16_X1Y0	RAMB16BWER	134	no color
28	bram1_instance/U0/xst_blk_mem_generator/gnativebmg.native_blk_mem_gen/valid.ctr/ramloop[1].ram.r/s6_noinit.ram/SP.SIMPLE_PRIM18.ram	RAMB16_X1Y2	RAMB16BWER	130	no color
29	clka	N4	IOB	1	no color

Figure 18: BRAM Components

Select one of the block RAM instances and select the Properties button  or “Edit” -> “Properties of Selected Item”. From the “Properties” window the block RAM instance name can be copied and pasted into the BMM file so that there are no inconsistencies in the name. Please see the example below of a block RAM instance name:

`bram0_instance/U0/xst_blk_mem_generator/gnativebmg.native_blk_mem_gen/valid.ctr/ramloop[0].ram.r/s6_noinit.ram/SP.SIMPLE_PRIM18.ram`

PlanAhead

To open the PlanAhead tool in the ISE software, select **“Tools” -> “PlanAhead” -> “I/O Pin Planning (PlanAhead) Pre/Post-Synthesis”**. The PlanAhead software opens as shown in Figure 19.

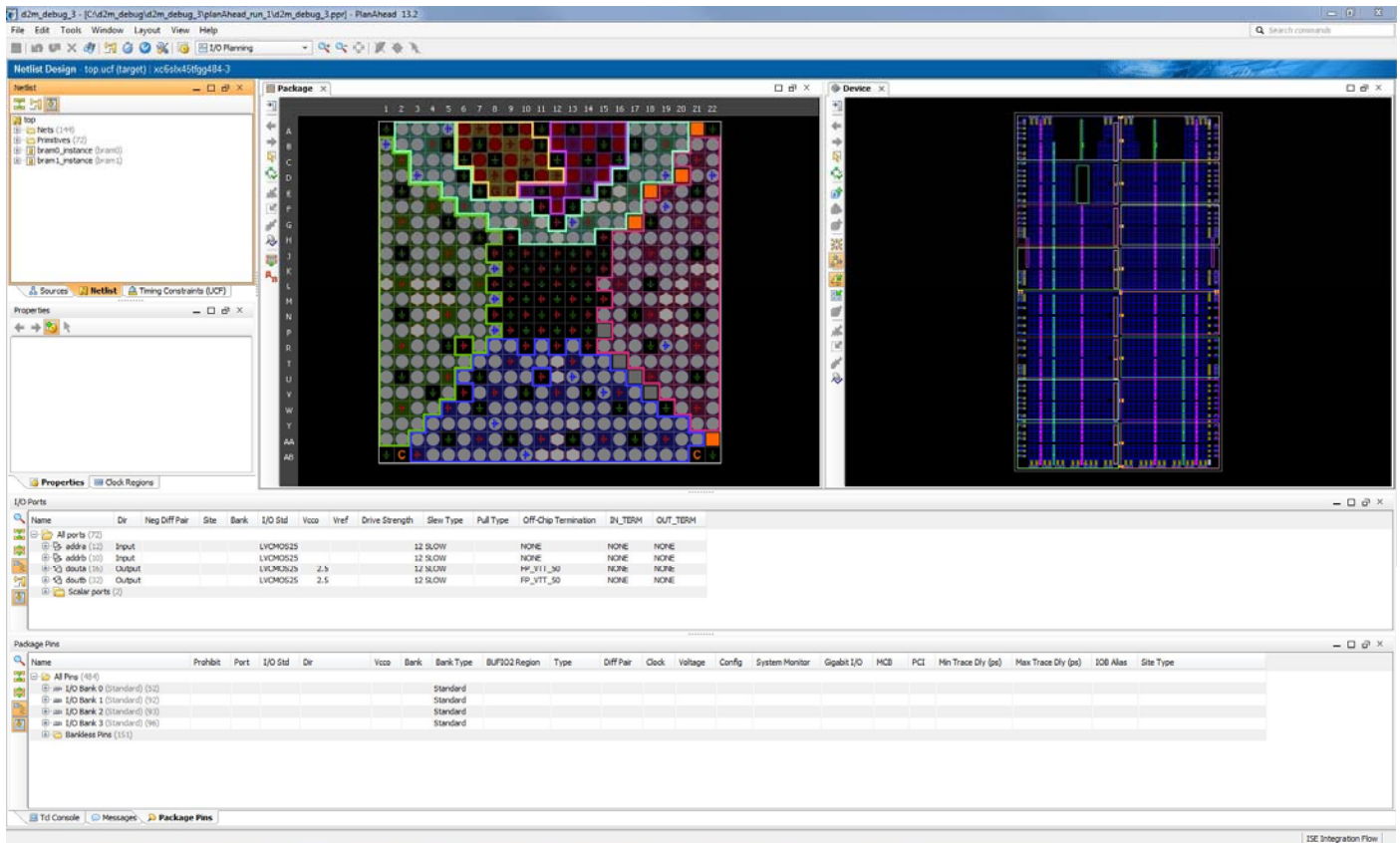


Figure 19: PlanAhead

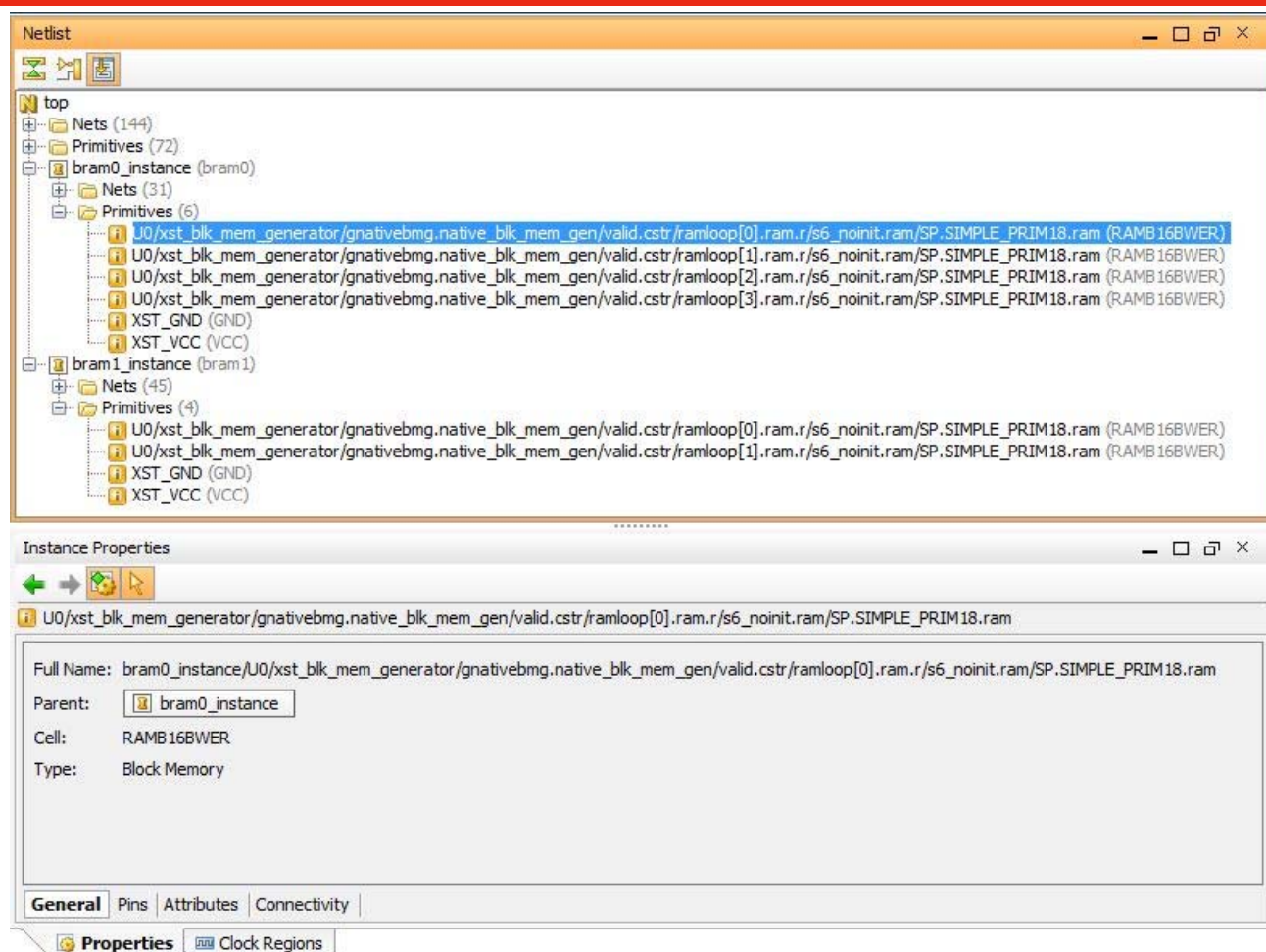


Figure 20: Netlist and Instance Properties

Data2MEM Examples

This section provides Data2MEM examples to illustrate how the content of the BMM file and the MEM file co-relates to the content of the Data2MEM output BIT file.

Example-1

The representation of the BMM file, for this example, is as follows:

```
ADDRESS_SPACE BRAM_0 COMBINED [0x00000000:0x00002FFF]
ADDRESS_RANGE RAMB16 /* 0x00000000 - 0x00001FFF */
BUS_BLOCK
  bram0_instance/U0/./ramloop[0].ram.r/./SP.SIMPLE_PRIM18.ram [3:0] PLACED = X0Y0;
  bram0_instance/U0/./ramloop[1].ram.r/./SP.SIMPLE_PRIM18.ram [7:4] PLACED = X0Y12;
  bram0_instance/U0/./ramloop[2].ram.r/./SP.SIMPLE_PRIM18.ram [11:8] PLACED =
  X0Y14;
  bram0_instance/U0/./ramloop[3].ram.r/./SP.SIMPLE_PRIM18.ram [15:12] PLACED =
  X0Y16;
END_BUS_BLOCK;
END_ADDRESS_RANGE;
ADDRESS_RANGE RAMB16 /* 0x00002000 - 0x00002FFF */
BUS_BLOCK
  bram1_instance/U0/./ramloop[0].ram.r/./SP.SIMPLE_PRIM18.ram [15:0] PLACED = X1Y0;
  bram1_instance/U0/./ramloop[1].ram.r/./SP.SIMPLE_PRIM18.ram [31:16] PLACED =
  X1Y2;
END_BUS_BLOCK;
END_ADDRESS_RANGE;
END_ADDRESS_SPACE;
```



In this example, if the MEM file is generated to store repeated DEADBEEF ABBA1234 value starting at 0x00000000, bram0 will be addressed and filled before bram1. How the data is stored is directly dependent on how the BMM file is generated and the size and configuration of the memory blocks. The following shows the content of the MEM file for this example.

MEM File:

```
@00000000  
DEADBEFF  
ABBA1234  
DEADBEFF  
ABBA1234  
  
|  
|  
|  
  
'Repeated Multiple Times until all ramloops are full'  
  
|  
|  
  
DEADBEFF  
ABBA1234  
DEADBEFF  
ABBA1234
```

Data2MEM Command:

```
data2mem -bm test bmm.bmm -bt test top.bit -bd test mem.mem -o b download.bit
```

Output Text File Dump Command:

```
data2mem -bm test bmm.bmm -bt download.bit -d > dump test.txt
```

Following is from the dump test.txt where it shows data being stored in the corresponding locations of the BRAMs:

```

    bram0 Ramloop[0] [0:3]
      00000000:   DB A1 DB A1 DB A1 DB A1 DB A1 DB A1 DB A1 DB A1 DB A1 DB A1 DB A1 DB A1
bram0 Ramloop[1] [4:7]
      00000000:   EE B2 EE B2 EE B2 EE B2 EE B2 EE B2 EE B2 EE B2 EE B2 EE B2 EE B2 EE B2
bram0 Ramloop[2] [8:11]
      00000000:   AE B3 AE B3 AE B3 AE B3 AE B3 AE B3 AE B3 AE B3 AE B3 AE B3 AE B3 AE B3
bram0 Ramloop[3] [12:15]
      00000000:   DF A4 DF A4 DF A4 DF A4 DF A4 DF A4 DF A4 DF A4 DF A4 DF A4 DF A4 DF A4
.....
.....

    bram1 Ramloop[0] [0:15]
      00000000:   DE AD AB BA DE AD AB BA DE AD AB BA DE AD AB BA DE AD AB BA 00 00 00 00 00 00
bram1 Ramloop[1] [16:31]
      00000000:   BE EF 12 34 BE EF 12 34 BE EF 12 34 BE EF 12 34 BE EF 12 34 BE EF 12 34 00 00 00 00 00 00 00 00

```

When the data is combined, it can easily be seen how it is being stored in the BRAMs:

bram0	Ramloop[0]	[0:3]	-	DB	A1	DB	A1
bram0	Ramloop[1]	[4:7]	-	EE	B2	EE	B2
bram0	Ramloop[2]	[8:11]	-	AE	B3	AE	B3
bram0	Ramloop[3]	[12:15]	-	DF	A4	DF	A4
bram1	Ramloop[0]	[0:15]	-	DE	AD	AB	BA
bram1	Ramloop[1]	[16:31]	-	BE	EF	12	34

If you want to address bram1 without having to wait for bram0 to fill, first you can address this separately in the MEM file by addressing 0x00002000 with the data required.

Example-2

This example shows a different memory configuration in the BMM file than the one show in Example-1.

```
ADDRESS SPACE BRAM_0 COMBINED [0x00000000:0x00002FFF]
  ADDRESS_RANGE RAMB16 /* 0x00000000 - 0x00001FFF */
    BUS_BLOCK
      bram0_instance/U0/./././ramloop[0].ram.r/./SP.SIMPLE_PRIM18.ram [15:0] PLACED = X0Y0;
    END_BUS_BLOCK;
  END_ADDRESS_RANGE;
  ADDRESS_RANGE RAMB16
    BUS_BLOCK
      bram0_instance/U0/./././ramloop[1].ram.r/./SP.SIMPLE_PRIM18.ram [15:0] PLACED = X0Y12;
    END_BUS_BLOCK;
  END_ADDRESS_RANGE;
  ADDRESS_RANGE RAMB16
    BUS_BLOCK
      bram0_instance/U0/./././ramloop[2].ram.r/./SP.SIMPLE_PRIM18.ram [15:0] PLACED = X0Y14;
    END_BUS_BLOCK;
  END_ADDRESS_RANGE;
  ADDRESS_RANGE RAMB16
    BUS_BLOCK
      bram0_instance/U0/./././ramloop[3].ram.r/./SP.SIMPLE_PRIM18.ram [15:0] PLACED = X0Y16;
    END_BUS_BLOCK;
  END_ADDRESS_RANGE;
  ADDRESS_RANGE RAMB16 /*0x00002000 - 0x00002FFF */
    BUS_BLOCK
      bram1_instance/U0/./././ramloop[0].ram.r/./SP.SIMPLE_PRIM18.ram [15:0] PLACED = X1Y0;
      bram1_instance/U0/./././ramloop[1].ram.r/./SP.SIMPLE_PRIM18.ram [31:16] PLACED = X1Y2;
    END_BUS_BLOCK;
  END_ADDRESS_RANGE;
END_ADDRESS_SPACE;
```

As bram0 is divided into separate bus blocks, the data will be stored into bram0 ramloop[0] first, and then bram0 ramlopp[1], and so on. For this example, the BRAM blocks will be addressed in the MEM file as shown below.

```
@00000000
DEADBEEF
ABBA1234
@000007FF
DEADBEEF
ABBA1234
@00000FFF
DEADBEEF
ABBA1234
@000017FF
DEADBEEF
ABBA1234
@00002000
DEADBEEF
ABBA1234
```

Data2MEM Command:

```
data2mem -bm test_bmm.bmm -bt test_top.bit -bd test_mem.mem -o b download.bit
```

Output Text File Dump Command:

```
data2mem -bm test bmm.bmm -bt download.bit -d > dump test.txt
```

Following is from the dump test.txt where it shows data being stored in the corresponding locations of the BRAMs:

[illegible]

The following shows the data stored in the block RAMs.

bram0	Ramloop[0]	[15:0]	-	DE	AD	BE	EF	AB	BA	12	34
bram0	Ramloop[1]	[15:0]	-	DE	AD	BE	EF	AB	BA	12	34
bram0	Ramloop[2]	[15:0]	-	DE	AD	BE	EF	AB	BA	12	34
bram0	Ramloop[3]	[15:0]	-	DE	AD	BE	EF	AB	BA	12	34
bram1	Ramloop[0]	[0:15]	-	DE	AD	AB	BA				
bram1	Ramloop[1]	[16:31]	-	BE	EF	12	34				

Example-3

This is another example of using Data2MEM. All steps have been described from the start for further clarity.

1. Generate Block Memory of 32 bits data width using Block Memory Generator in the CORE Generator software.
2. Instantiate the generated block memory in the top level VHDL file as shown below:

```
entity top is
  Port ( clka : in  STD_LOGIC;
        addra : in  STD_LOGIC_VECTOR (7 downto 0);
        douta : out STD_LOGIC_VECTOR (31 downto 0));
end top;

architecture Behavioral of top is

  COMPONENT block_memory_v4
  PORT(
    clka: IN std_logic;
    addra: IN std_logic_VECTOR(7 downto 0);
    douta: OUT std_logic_VECTOR(31 downto 0));
  END COMPONENT;

begin

  v4_block_memory: block_memory_v4 PORT MAP(
    clka => clka,
    addra => addra,
    douta => douta
  );

end Behavioral;
```

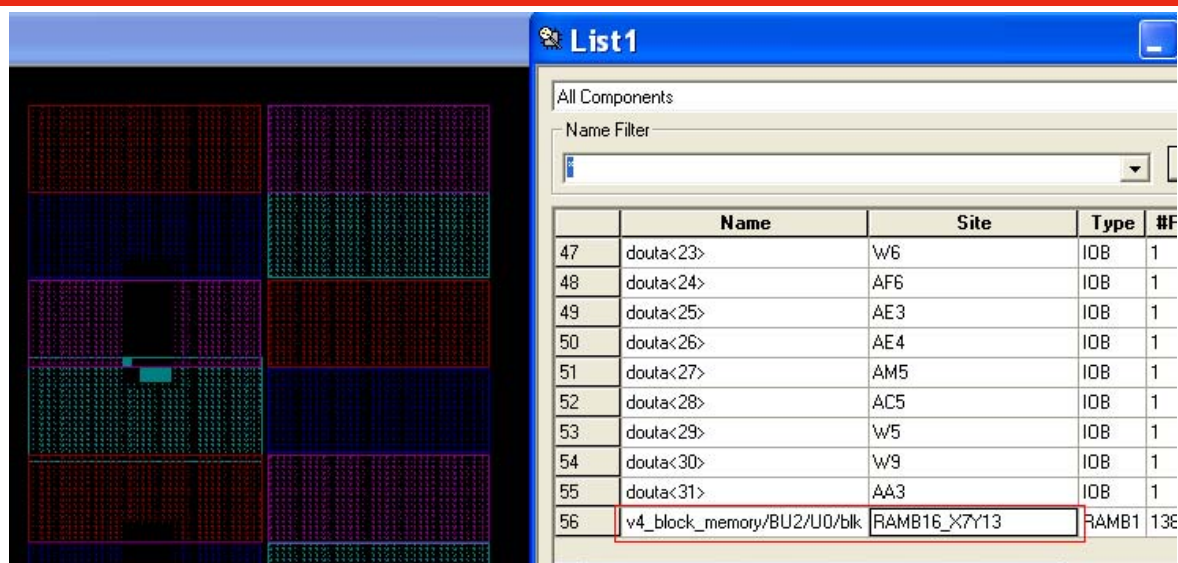
3. The next step is to create a BMM file. The BMM file for this example is shown below:

```
ADDRESS_SPACE v4_minor RAMB16 [0x00000000:0x000007FF]
  BUS_BLOCK
    v4_block_memory/BU2/U0/blk_mem_generator/valid.cstr/ramloop[0].ram.r/v4_noinit.ra
m/SP.WIDE_PRIM.SP [31:0] PLACED = X7Y13;
  END_BUS_BLOCK;
END_ADDRESS_SPACE;
```

There are two main questions to address while writing a BMM file.

- a. How do you find the block RAM location?
- b. How do you calculate the address range?

For question 'a', as discussed earlier, open FPGA Editor.



Highlight the BRAM in the red box above and copy the memory location as shown below:

```
site "RAMB16_X7Y13", type = RAMB16 (RPM grid X253Y104)
site "RAMB16_X7Y13", type = RAMB16 (RPM grid X253Y104)
site "RAMB16_X7Y13", type = RAMB16 (RPM grid X253Y104)
<RAMB16>; bel <v4_block_memory/BU2/U0/blk_mem_generator/valid.cstr/ramloop[0].ram.r/v4_noinit.ram/SP.WIDE_PRIM.SP>.
```

The question in 'b' was how to calculate the address range:

`ADDRESS_SPACE v4_minor RAMB16 [0x00000000:0x000007FF]`

There is only one memory block used in this example. The size of the memory block is 16Kbit. Since the data width selected in the block memory CORE Generator interface was 32, the memory will be arranged as shown in Figure 21:

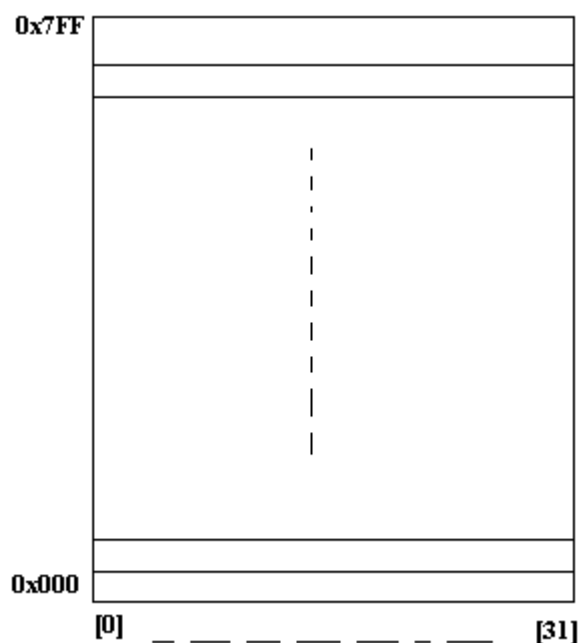


Figure 21 - Memory Address Range

There are 512 lines in the above memory with 4 bytes (32 bits) in each line. The total size of the memory is $512 \times 32 = 16384$ bits = 16Kbit.

4. The content of the mem file is shown below. This initializes the first 16bytes of the memory.

```
@0000
00000000
@4
DEADBEEF
@8
DEADABBA
@C
DEADFEED
```

5. You now have all three required files to run data2mem. The following data2mem command line is used to initialize the block memory with the MEM file described in '4' above.

```
data2mem -bm top_bd.bmm -bt top.bit -bd test.mem tag v4_minor -o b top_mem.bit
```

6. The output BIT file top_mem.bit will have its block RAM initialized with the MEM file. To verify whether this has been done or not, the bit file can be dumped using the following command:

```
data2mem -bm top_bd.bmm -bt top_mem.bit -d > dump_mem.txt
```

Open dump_mem.txt and search for `'blk_mem_generator/valid.cstr/ramloop[0].ram'`. This is a part of the memory path in step '3' above.

```
BRAM data, Column 07, Row 13. Design instance "v4_block_memory/BU2/
U0/blk_mem_generator/valid.cstr/ramloop[0].ram.r/v4_noinit.ram/SP.WIDE_PRIM.SP".
00000000: 00 00 00 00 DE AD BE EF DE AD AB BA DE AD FE ED 00 00 00 00 00 00 00 00
00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Data2Mem Verification in EDK

As mentioned earlier, Data2MEM is used in EDK to initialize block RAM. In an EDK project, if the software application is stored in block RAM, the EDK tool invokes Data2MEM to initialize the block RAM in system.bit file, generated after running 'Generate Bitstream'. The output bit file from Data2MEM is download.bit file which would have the block RAMs initialized with the software application. This section describes how to verify whether the block RAM in an EDK design has been correctly initialized or not.

The design used here is a BSB (Base System Builder) design for an ML505 board. The software application is allocated in the block RAMs. The remainder of this section describes steps to verify the initialization of block RAMs in an EDK design.

1. Implement the EDK design. This will give you 'system.bit' file.
2. Using Linker Script, allocate software sections in block RAMs.
3. Mark your application to be initialized in block RAMs. By doing so, when you invoke 'Update Bitstream' the sections defined in the linker script will be initialized into corresponding block RAM sections as defined by the linker script.
4. Click on **Device Configuration** -> **Update Bitstream**. This will generate the 'download.bit' file in the 'implementation' folder of your project directory.
5. Dump your elf file. The dump of your elf file will give you the location of your instruction code in the corresponding memory address of your block RAMs. Next, dump the bit file as well. The bit file dump will show you how the block RAMs have been initialized. It will be initialized with your software instruction code.

The instruction code in the elf file dump for a particular memory address should be the same as the bit file dump for the same address.

One more step of verification can be performed by reading back the data from the hardware from that particular memory location via XMD. The value read back should be the same as in the dump of the elf and the bit file. On the other hand, you can also write a piece of code to read back the same memory location at runtime. The output that will be displayed on the HyperTerminal should be the same as the one you get from the elf and the bit file dump. All of this is explained below. The following command line shows how to dump an elf file.

➤ Mb-objdump -S TestApp_Memory/executable.elf > elf_dump.txt

- The following code snippet is from elf_dump.txt. The software instruction code is placed in the block RAM starting from the base address of the block RAM assigned in the MHS file.

```
87388000 <_start1>:
87388000:      31a00588      addik    r13, r0, 1416
87388004:      30400428      addik    r2, r0, 1064
87388008:      30200da0      addik    r1, r0, 3488
8738800c:      b9f400c0      brlid    r15, 192
87388010:      80000000      or       r0, r0, r0
87388014:      b9f408a8      brlid    r15, 2216
87388018:      30a30000      addik    r5, r3, 0
```

```
BEGIN xps_bram_if_cntlr
PARAMETER INSTANCE = xps_bram_if_cntlr_1
PARAMETER HW_VER = 1.00.a
PARAMETER C SPLB NATIVE DWIDTH = 32
PARAMETER C BASEADDR = 0x87388000
PARAMETER C HIGHADDR = 0x87389fff
BUS_INTERFACE SPLB = mb_plb
BUS_INTERFACE PORTA = xps_bram_if_cntlr_1_port
END
```

- Dump the download.bit file as shown below:

```
data2mem -bm system_bd.bmm -bt download.bit -d > dump.txt
@pause
```

- Open the system_bd.bmm file. This file is found in the 'implementation' directory.

```
////////////////////////////////////
//
// Processor 'microblaze_0' address space 'xps_bram_if_cntlr_1_bram_combined'
// 0x87388000:0x87389FFF (8 KB).
//
////////////////////////////////////

ADDRESS_SPACE xps_bram_if_cntlr_1_bram_combined RAMB32 [0x87388000:0x87389FFF]
BUS_BLOCK
  xps_bram_if_cntlr_1_bram/xps_bram_if_cntlr_1_bram/ramb36_0 [31:16] PLACED = X1Y6;
  xps_bram_if_cntlr_1_bram/xps_bram_if_cntlr_1_bram/ramb36_1 [15:0] PLACED = X1Y18;
END_BUS_BLOCK;
END_ADDRESS_SPACE;

END_ADDRESS_MAP;
```

As seen here, xps_bram starts at address 0x87388000. This BRAM is physically located at BRAM locations X1Y6 and X1Y18. The next task is to find the content of the BRAMs ramb36_1 [15:0] and ramb36_0 [31:16] in the dump of the download.bit file. The content of these two BRAMs will give the instruction code located at address 0x87388000.

- Open dump.txt and search for xps_bram, you should see the following two separate sections.

```

BRAM data, Column 01, Row 18. Design instance "xps_bram_if_cntlr_1_bram/xps_bram_if_cntlr_1_bram/ramb36_1".
00000000: 05 BB 04 28 0D A0 00 00 00 00 08 A8 00 00 00 00 05 88 FF E4 00 00 00 14 00 40 04 30 20 00 00 00
00000020: 04 30 00 00 FF EC 00 04 00 00 00 00 00 10 05 8D 18 00 00 00 00 01 05 88 00 00 00 08 00 1C 00 00
00000040: 00 00 FF E4 00 00 05 80 05 8C 00 0C 18 00 00 00 05 84 00 00 00 00 14 05 84 00 0C 20 00 00 00
00000060: 00 00 00 08 00 1C FF EC 00 00 05 88 05 88 38 00 00 14 00 00 00 04 38 00 FF F4 05 88 05 AC 38 00
00000080: 00 14 00 00 00 04 38 00 FF F4 06 54 00 00 0B D0 00 00 00 00 00 00 00 00 00 00 2C 00 00 00 0B E0 00 00

BRAM data, Column 01, Row 06. Design instance "xps_bram_if_cntlr_1_bram/xps_bram_if_cntlr_1_bram/ramb36_0".
00000000: 31 A0 30 40 30 20 B9 F4 80 00 B9 F4 30 A3 B8 00 E0 60 30 21 F9 E1 BC 03 B8 00 F8 60 99 FC 80 00
00000020: E8 60 E8 83 BE 24 30 63 B0 00 30 60 BC 03 30 A0 99 FC 80 00 30 60 F0 60 E9 E1 B6 0F 30 21 B0 00
00000040: 30 60 30 21 F9 E1 30 A0 30 C0 BC 03 99 FC 80 00 E8 60 B0 00 30 80 BC 03 30 A0 BC 04 99 FC 80 00
00000060: E9 E1 B6 0F 30 21 20 21 F9 E1 20 C0 20 E0 06 46 BC 72 F8 06 20 C6 06 46 BC 92 20 C0 20 E0 06 46
00000080: BC 72 F8 06 20 C6 06 46 BC 92 B9 F4 80 00 B9 F4 80 00 20 C0 20 E0 B9 F4 20 A0 32 63 B9 F4 80 00
000000A0: B9 F4 80 00 C9 E1 30 73 B6 0F 20 21 30 21 F9 E1 30 A0 B0 00 E8 C0 B9 F4 80 00 E9 E1 10 60 B6 0F

```

As discussed in Step 8, the content inside the red boxes above is actually the content of the block RAMs at its base address location. This value is the same as what you read from the elf file dump as described in Step 6.

- Now, let's read back from the hardware itself in XMD at address 0x87388000. This can be done using the mrd command. This is shown in Figure 22.

```

Hard Multiplier Support.....on - (Mul32)
Barrel Shifter Support.....off
MSR clr/set Instruction Support....on
Compare Instruction Support.....on

Connected to MDM UART Target
Connected to "mb" target. id = 0
Starting GDB server for "mb" target <id = 0> at TCP por
XMD> mrd 0x87388000 10
87388000: 31A00588
87388004: 30400428
87388008: 30200DA0
8738800C: B9F400C0
87388010: 80000000
87388014: B9F408A8
87388018: 30A30000
8738801C: B8000000
87388020: E0600588
87388024: 3021FFE4

```

Figure 22 - Memory Read in XMD

11. As discussed in Step 5, you can also verify by reading the memory location in your C code. The memory content will be displayed on the HyperTerminal. For this, include the following code in your software application.

```
#include "xparameters.h"

#include "stdio.h"

#include "xutil.h"

//=====

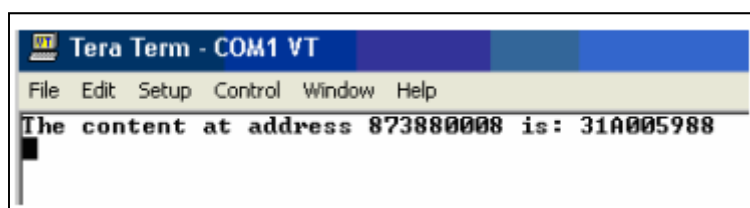
int main (void) {

    Xuint32 *bram_base_addr;
    bram_base_addr = (Xuint32 *)0x87388000;

    xil_printf("The content at address %0x8 is: %0x8 \r\n",
        bram_base_addr, *(bram_base_addr));

    return 0;
}
```

12. When you download the download.bit file, you should see the following output:



Conclusion

This document presented a conceptual background on using Data2MEM. Also, a detailed explanation was given on how to create a BMM file, MEM file and how to verify the content of the block RAM in the BIT file generated by Data2MEM. This is useful especially in debugging designs when you are not getting the correct output while reading the content of the block RAM instantiated in your design.

If this document does not help resolve your problem, please open a WebCase with Xilinx Technical Support and include the details of your investigation.

References

1. **Data2Mem User Guide**,
http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/data2mem.pdf

Revision History

04/30/2012 - Initial Release