Data2MEM User Guide

UG658 (v 11.2) June 24, 2009

XILINX®

Xilinx Trademarks and Copyright Information

Xilinx is disclosing this user guide, manual, release note, and/or specification (the "Documentation") to you solely for use in the development of designs to operate with Xilinx hardware devices. You may not reproduce, distribute, republish, download, display, post, or transmit the Documentation in any form or by any means including, but not limited to, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Xilinx. Xilinx expressly disclaims any liability arising out of your use of the Documentation. Xilinx reserves the right, at its sole discretion, to change the Documentation without notice at any time. Xilinx assumes no obligation to correct any errors contained in the Documentation, or to advise you of any corrections or updates. Xilinx expressly disclaims any liability in connection with technical support or assistance that may be provided to you in connection with the Information.

THE DOCUMENTATION IS DISCLOSED TO YOU "AS-IS" WITH NO WARRANTY OF ANY KIND. XILINX MAKES NO OTHER WARRANTIES, WHETHER EXPRESS, IMPLIED, OR STATUTORY, REGARDING THE DOCUMENTATION, INCLUDING ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT OF THIRD-PARTY RIGHTS. IN NO EVENT WILL XILINX BE LIABLE FOR ANY CONSEQUENTIAL, INDIRECT, EXEMPLARY, SPECIAL, OR INCIDENTAL DAMAGES, INCLUDING ANY LOSS OF DATA OR LOST PROFITS, ARISING FROM YOUR USE OF THE DOCUMENTATION.

Table of Contents

Introduction to Data2MEM

The *Data2MEM User Guide* describes how Data2MEM automates and simplifies setting the contents of Block RAM memory on Xilinx® FPGA family products.

This chapter includes:

Data2MEM Overview

Data2MEM is a data translation tool for contiguous blocks of data across multiple block RAMs which constitute a contiguous logical address space. With the combination of Virtex® series devices and an embedded CPU on a single chip, Data2MEM incorporates CPU software images into FPGA bitstreams. As a result, CPU software can be executed from block RAM-built memory within a FPGA bitstream. This provides a powerful and flexible means of merging parts of CPU software and FPGA design tool flows. Data2MEM also provides a simplified means for initializing block RAMs for non-CPU designs.

Data2MEM automates a process to a simplified technique, and also:

- Minimizes the effect of existing tool flows for both FPGA and CPU software designers.
- Limits the time delay one tool flow imposes on another for testing changes or fixing verification problems.
- Isolates the process to a minimal number of steps.
- Reduces or eliminates the requirement for one tool flow user (for example, a CPU software or FPGA designer) to learn the other tool flow steps and details.

Data2MEM is supported on the following platforms:

- Linux
- Windows XP
- Windows Vista

Data2MEM Features

Data2MEM supports the following devices:

• Virtex®-4

• Virtex-5

• Spartan®-3A

• Spartan-3AN

• Spartan-3ADSP

Data2MEM includes the following features:

• Reads a new block RAM Memory Map (BMM) file that contains a textual syntax describing arbitrary arrangements of block RAM usage and depth. This syntax also includes CPU bus widths and bit lane interleaving.

• Adapts to multiple data widths available from block RAM models.

• Reads Executable and Linkable Format (ELF) files, or DWARF Debugging Information Format (DRF) files as input for CPU software code images. No changes are required from any third party CPU software tools to translate CPU software code from its natural file format.

• Reads MEM format text files as input for block RAM contents. The text format can be either hand or machine generated.

• Optionally produces formatted text dumps of BIT, ELF, and DRF files

• Produces Verilog and VHDL for initialization files for pre- and post-synthesis simulation.

• Integrates initialization data into post-place and route (post-PAR) simulations.

• Produces MEM files for Verilog simulations with third-party memory models.

• Replaces the contents of block RAM in BIT files directly, without intervention of another implementation tool, thus avoiding lengthy implementation tool runs.

• Invokes as either a command line tool or as an integrated part of the Xilinx® implementation tool flow.

• Recognizes common text line ending types (such as Windows and Linux) and uses them interchangeably.

• Allows the free-form use of `//` and `/*...*/` commenting syntax in text input files.

Uses for Data2MEM

You can use Data2MEM for the following processes:

1. In software design, as a command line tool for generating updated BIT files. For more information, see, Using the Command Line.

2. In hardware design, to integrate Data2MEM with the Xilinx® implementation tools. For more information, see Using Integrated ISE® Design Suite Implementation Tools.

3. As a command line tool to generate behavioral simulation files.

The figure below represents how the two tool flows operate within discrete-chip CPU and FPGA designs: two separate source bases, bit images, and boot mechanisms.

When integrating discrete-chip CPU and FPGA designs into a single FPGA chip, the source bases can remain separated, which means the portion of the tool flows that operate on sources can also remain separated.

However, a single FPGA chip implies a single boot image, which must contain the merged CPU and FPGA bit images. The tight integration of CPU and FPGA requires closer coupling within the FPGA simulation process. To produce combined bit images, Data2MEM combines the CPU and FPGA tool flow outputs, while leaving the two flows unchanged.

The following figure shows a high-level tool flow for CPU software and an FPGA design.

High Level Software and Hardware Tool Flows
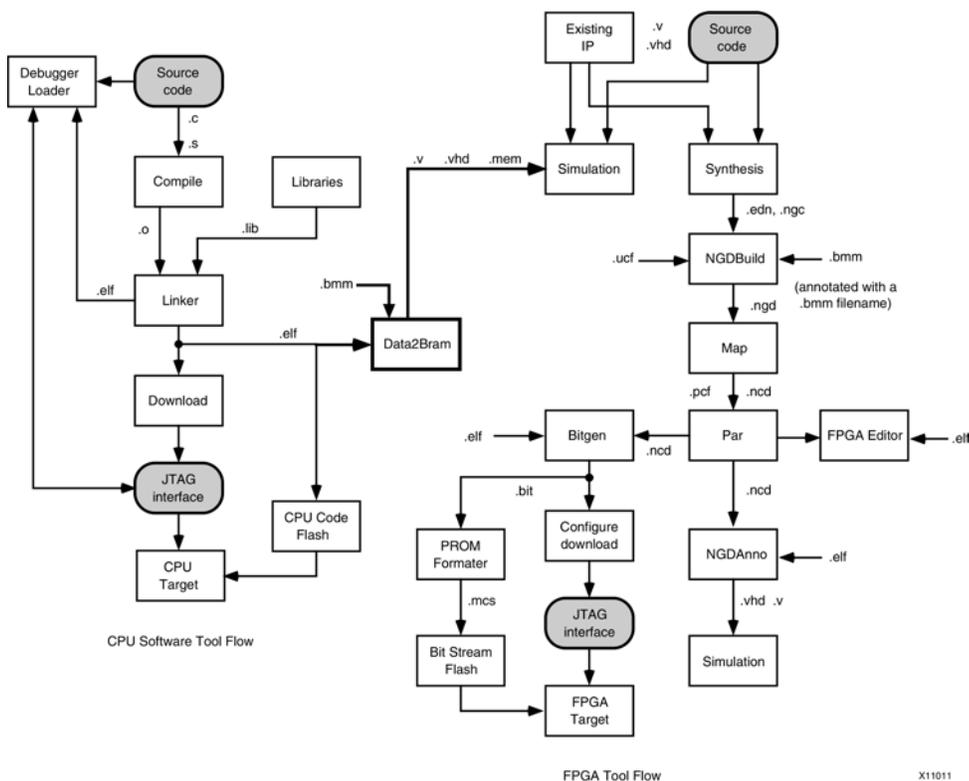
The following sections describe the CPU software source code and FPGA design data flow.

CPU Software Source Code and FPGA Source Code

The following sections describe how Data2MEM uses CPU software source code and FPGA design code:

- CPU Software Source Code
- FPGA Source Code

CPU Software Source Code

CPU software source code is used in the form of high-level C files and assembly-level Sfiles. These files are compiled into object (.o) link files. The object files, with prebuilt object generated libraries, are linked together into a single executable code image.

The output of the linker is an ELF file. The ELF contents can be downloaded via JTAG, stored in non-volatile memory, or stored in the BIT file if it is small enough.

CPU Software Tool Flow
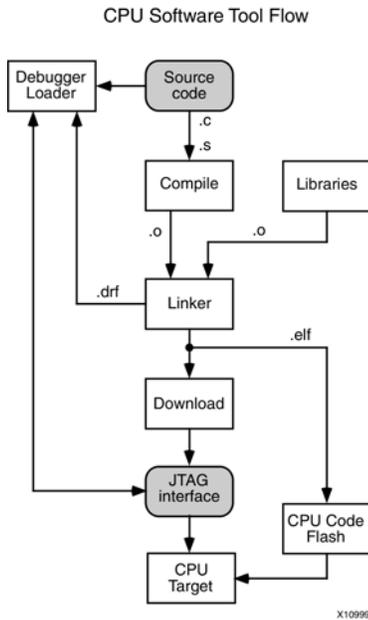
FPGA Source Code

The FPGA source code is used in the form of V, VHD, and Electronic Data Interchange Format (EDIF) files. These files pass through the design flow as follows:

- The files are used in various styles of hardware simulation or the files are synthesized into EDN or NGC intermediate files.

- A User Constraints File (UCF) and the intermediate EDN or NGC file are run through NGDBuild, MAP, and Place and Route (PAR) to produce a Native Circuit Description (NCD) file.

  Note NGDBuild is a program that converts all input design netlists and then writes the results into a single merged file.

- Bitgen converts the NCD file into an FPGA Bitstream (BIT) file that can be used to configure the FPGA.

- The BIT file can be downloaded to the FPGA directly or programmed into the FPGA boot configure flash.

Data2MEM Design Considerations

This section summarizes the design factors necessary for mapping CPU software code to block RAM-implemented address spaces.

The following flow represents a logical layout and grouping of block RAM memory only. FPGA logic must be constructed to translate CPU address requests into physical block RAM selection. The design of FPGA logic is not covered in this Guide.

Following are design considerations for block RAM-implemented address spaces:

- The block RAMs come in fixed-size widths and depths, where CPU address spaces might need to be much larger in width and depth than a single block RAM. Consequently, multiple block RAMs must be logically grouped together to form a single CPU address space.

- A single CPU bus access is often multiple bytes wide of data, for example, 32 or 64 bits (4 or 8 bytes) at a time.

- CPU bus accesses of multiple data bytes might also access multiple block RAMs to obtain that data. Therefore, byte-linear CPU data must be interleaved by the bit width of each block RAM and by the number of block RAMs in a single bus access. However, the relationship of CPU addresses to block RAM locations must be regular and easily calculable.

- CPU data must be located in a block RAM-constructed memory space relative to the CPU linear addressing scheme, and not to the logical grouping of multiple block RAMs.

- Address space must be contiguous, and in whole multiples of the CPU bus width. Bus bit lane interleaving is allowed only in the sizes supported by Virtex® device block RAM port sizes. The data bus sizes by device family are:

  - 1, 2, 4, 8, 9, 16, 18, 32, and 36 bits for Spartan®-3 devices and Virtex-4 devices.

  - 1, 2, 4, 9, 18, 36, and 72 bits for Virtex-5 devices. For more information, see Block RAM Configurations by Device Family.

    Note When using parity, Data2MEM assumes the parity bits occupy the upper (Most Significant) bits of the device data bus. For more information, see Bit Lane Definitions (Memory Device Usage)

- Addressing must account for the differences in instruction and data memory space. Because instruction space is not writable, there are no address width restrictions. However, data space is writable and usually requires the ability to write individual bytes. For this reason, each bus bit lane must be addressable.

- The size of the memory map and the location of the individual block RAMs affect the access time. Evaluate the access time after implementation to verify that it meets the design specifications.

  For more information, see High Level Software and Hardware Tool Flows.

- A 16 Kbyte address space from CPU address `0xFFFFC000` to `0xFFFFFFFF`, constructed from the logical grouping of 32 4-Kbit block RAMs.

- Each block RAM is configured to be 8 bits wide, and 512 bytes deep.

- CPU bus accesses are 8 block RAMs (64 bits) wide, with each column of block RAMs occupying an 8-bit wide slice of a CPU bus access called a *Bit Lane*.

- Each row of 8 block RAMs in a bus access are grouped together in a *Bus Block*. Hence, each Bus Block is 64 bits wide and 4096 bytes in size.

- The entire collection of block RAMs is grouped together into a contiguous address space called an Address Block.

Example Block RAM Address Space Layout
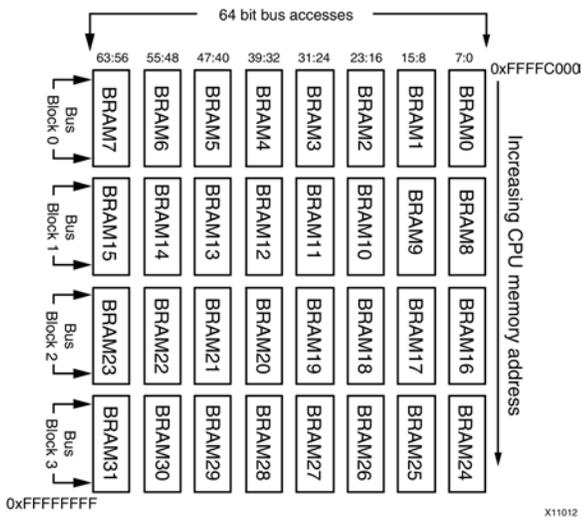
The address space in the figure above consists of four bus blocks. The upper right corner address is `0xFFFFC000`, and the lower left corner address is `0xFFFFFFFF`. Because a bus access obtains 8 data bytes across 8 block RAMs, byte-linear CPU data must be interleaved by 8 bytes in the block RAMs.

In this example:

- Byte 0 goes into the first byte location of bit lane block RAM7, byte 1 goes into the first byte location of Bit Lane block RAM6; and so forth, to byte 7.

- CPU data byte 8 goes into the second byte location of Bit Lane block RAM7, byte 9 goes into the second byte location of Bit Lane block RAM6 and so forth, repeating until CPU data byte 15.

- This interleave pattern repeats until every block RAM in the first bus block is filled.

- This process repeats for each successive bus block until the entire memory space is filled, or the input data is exhausted.

As described in BMM File Syntax the order in which bit lanes and bus blocks are defined controls the filling order. For the sake of this example, assume that bit lanes are defined from left to right, and bus blocks are defined from top to bottom.

This process is called Bit Lane Mapping, because these formulas are not restricted to byte-wide data. This is similar, but not identical, to the process embedded software programmers use when programmed CPU code is placed into the banks of fixed-size EPROM devices.

The important distinctions to note between the two processes are:

- Embedded system developers generally use a custom software tool for byte lane mapping for a fixed number and organization of byte-wide storage devices. Because the number and organization of the devices cannot change, these tools assume a specific device arrangement. Consequently, little or no configuration options are provided. By contrast, the number and organization of FPGA block RAMs are completely "soft" (within FPGA limits). Ay tool for byte lane mapping for block RAMs must support a large set of device arrangements.

- Existing byte lane mapping tools assume an ascending order of the physical addressing of byte-wide devices because that is how board-level hardware is built. By contrast, FPGA block RAMs have no fixed usage constraints and can be grouped together with block RAMs anywhere within the FPGA fabric. Although this example displays block RAMs in ascending order, block RAMs can be configured in any order.

- Discrete storage devices are typically only one or two bytes (8 or 16 bits) wide, or occasionally, four bits wide. Existing tools often assume that storage devices are a single width. Virtex-4 device and Virtex-5 device block RAM, however, can be configured in several widths, depending on hardware design requirements. The tables in Block RAM Configurations by Device Family specify the block RAM widths.

- Since existing tools have limited configuration needs, a simple command line interface will usually suffice. The block RAM usage adds more complexity, and requires human-readable syntax to describe the mapping between address spaces and block RAM utilization.

Block RAM Configurations by Device Family

This section discusses Block RAM Configurations by Device Family and includes:

- Block RAM Configurations for Spartan®-3 Devices, Spartan-3A Devices, and Spartan-3E Devices
- Block RAM Configurations for Virtex®-4 Devices
- Block RAM Configurations for 18 Kbit Virtex-5 Devices
- Block RAM Configurations for 36 Kbit Virtex-5 Devices

Block RAM Configurations for Spartan-3 Devices, Spartan-3A Devices, and Spartan-3E Devices

Primitive	Data Depth	Data Width	Memory Type
RAMB16_S1	16384	1	RAMB16
RAMB16_S2	8192	2	RAMB16
RAMB16_S4	4096	4	RAMB16
RAMB16_S9	2048	8	RAMB16
RAMB16_S9	2048	9	RAMB18
RAMB16_S18	1024	16	RAMB16
RAMB16_S18	1024	18	RAMB18
RAMB16_S36	512	32	RAMB32
RAMB16_S36	512	36	RAMB36

Block RAM Configurations for Virtex-4 Devices

Primitive	Data Depth	Data Width	Memory Type
RAMB16	16384	1	RAMB16
RAMB16	8192	2	RAMB16
RAMB16	4096	4	RAMB16
RAMB16	2048	8	RAMB16
RAMB16	2048	9	RAMB18
RAMB16	1024	16	RAMB16
RAMB16	1024	18	RAMB18
RAMB16	512	32	RAMB32
RAMB16	512	36	RAMB36

Block RAM Configurations for 18 Kbit Virtex-5 Devices

Primitive	Data Depth	Data Width	Memory Type
RAMB18	16384	1	RAMB18
RAMB18	8192	2	RAMB18
RAMB18	4096	4	RAMB18
RAMB18	2048	8	RAMB16
RAMB18	2048	9	RAMB18
RAMB18	1024	16	RAMB16
RAMB18	1024	18	RAMB18
RAMB18SDP	512	32	RAMB32
RAMB18SDP	512	36	RAMB36

Block RAM Configurations for 36 Kbit Virtex-5 Devices

Primitive	Data Depth	Data Width	Memory Type
RAMB36	32768	1	RAMB32
RAMB36	16384	2	RAMB32
RAMB36	8192	4	RAMB32
RAMB36	4096	8	RAMB32
RAMB36	4096	9	RAMB36
RAMB36	2048	16	RAMB32
RAMB36	2048	18	RAMB36
RAMB36	1024	32	RAMB32
RAMB36SDP	512	64	RAMB32

Input and Output Files

This chapter describes each input and output file type, and how it is used or created by Data2MEM. This chapter includes:

- Block RAM Memory Map (BMM) Files
- Executable and Linkable Format (ELF)
- Memory (MEM) Files
- Bitstream (BIT) Files
- Verilog Files
- VHDL Files
- User Constraints File (UCF) Files

The following figure shows the range of files, and their input and output relationship to Data2MEM.
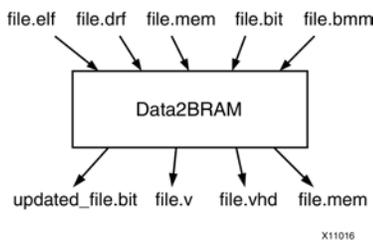
Block RAM Memory Map (BMM) Files

A Block RAM Memory Map (BMM) file is a text file that syntactically describes how individual block RAMs make up a contiguous logical data space. Data2MEM uses the BMM file as input to direct the translation of data into the proper initialization form.

You can create a BMM file:

- Manually
- By using Data2MEM to generate BMM file templates
- By means of automated scripting

You can customize the templates for a specific design.

Because it is a text file, you can edit a BMM file directly. BMM files support both `//` and `/*...*/` commenting styles.

For more information, see BMM Features and BMM File Syntax.

The Example BMM File below shows the text-based syntax used to describe the organization of block RAM usage in a flexible and readable form. The address space defined in the example is the same BMM definition as the address shown in *High Level Software and Hardware Tool Flows* in Uses for Data2MEM.

```
/*****************************************************
*
* FILE : example.bmm
*
* Define a BRAM map for the RAM controller memory space. The
* address space 0xFFFF0000 - 0xFFFFFFFF, 64k deep by 64 bits wide.
*
***************************************************** /

ADDRESS_SPACE ram_cntlr RAMB16 [0XFFFF0000:0XFFFFFFFF]

  // Bus access map for the lower 16k, CPU address 0xFFFF0000 - 0xFFFF3FFF
  BUS_BLOCK
    top/ram_cntlr/ram7 [63:56] LOC = R3C5;
    top/ram_cntlr/ram6 [55:48] LOC = R3C6;
    top/ram_cntlr/ram5 [47:40] LOC = R3C7;
    top/ram_cntlr/ram4 [39:32] LOC = R3C8;
    top/ram_cntlr/ram3 [31:24] LOC = R4C5;
    top/ram_cntlr/ram2 [23:16] LOC = R4C6;
    top/ram_cntlr/ram1 [15:8] LOC = R4C7;
    top/ram_cntlr/ram0 [7:0] LOC = R4C8;
  END_BUS_BLOCK;

  // Bus access map for next higher 16k, CPU address 0xFFFF4000 - 0xFFFF7FFF
  BUS_BLOCK
    top/ram_cntlr/ram15 [63:56] OUTPUT = ram15.mem;
    top/ram_cntlr/ram14 [55:48] OUTPUT = ram14.mem;
    top/ram_cntlr/ram13 [47:40] OUTPUT = ram13.mem;
    top/ram_cntlr/ram12 [39:32] OUTPUT = ram12.mem;
    top/ram_cntlr/ram11 [31:24] OUTPUT = ram11.mem;
    top/ram_cntlr/ram10 [23:16] OUTPUT = ram10.mem;
    top/ram_cntlr/ram9 [15:8] OUTPUT = ram9.mem;
    top/ram_cntlr/ram8 [7:0] OUTPUT = ram8.mem;
  END_BUS_BLOCK;

  // Bus access map for next higher 16k, CPU address 0xFFFF8FFF-0xFFFFBFFF
  BUS_BLOCK
    top/ram_cntlr/ram23 [63:56];
    top/ram_cntlr/ram22 [55:48];
    top/ram_cntlr/ram21 [47:40];
    top/ram_cntlr/ram20 [39:32];
    top/ram_cntlr/ram19 [31:24];
    top/ram_cntlr/ram18 [23:16];
    top/ram_cntlr/ram17 [15:8];
    top/ram_cntlr/ram16 [7:0];
  END_BUS_BLOCK;

  // Bus access map for next higher 16k, CPU address 0xFFFFC000 - 0xFFFFFFFF
  BUS_BLOCK
    top/ram_cntlr/ram31 [63:56];
    top/ram_cntlr/ram30 [55:48];
    top/ram_cntlr/ram29 [47:40];
    top/ram_cntlr/ram28 [39:32];
    top/ram_cntlr/ram27 [31:24];
    top/ram_cntlr/ram26 [23:16];
    top/ram_cntlr/ram25 [15:8];
    top/ram_cntlr/ram24 [7:0];
  END_BUS_BLOCK;

END_add_SPACE;
```

Executable and Linkable Format (ELF) Files

An Executable and Linkable Format (ELF) file is a binary data file that contains an executable CPU code image ready for running on a CPU. ELF files are produced by software compiler and linker tools. For more information on creating ELF files, see your software tool documentation.

Data2MEM uses ELF files as its basic data input form. Because ELF files contain binary data, they cannot be directly edited. Data2MEM provides some facilities for examining ELF file content.

For more information, see Using the Command Line and Using Integrated Implementation Tools.

Memory (MEM) Files

A Memory (MEM) file is a text file that describes contiguous blocks of data. MEM files can be edited directly. Data2MEM allows the free-form use of both `//` and `/*...*/` commenting styles.

Data2MEM uses MEM files for both data input and output.

The format of MEM files is an industry standard, which consists of two basic elements: a hexadecimal address specifier and hexadecimal data values. An address specifier is indicated by an `@` character followed by the hexadecimal address value. There are no spaces between the `@` character and the first hexadecimal character.

Hexadecimal data values follow the hexadecimal address value, separated by spaces, tabs, or carriage-return characters. Data values can consist of as many hexadecimal characters as desired. However, when a value has an odd number of hexadecimal characters, the first hexadecimal character is assumed to be a zero. For example, hexadecimal values:

```
A, C74, and 84F21
```

are interpreted as the values:

```
0A, 0C74, and 084F21
```

The common `0x` hexadecimal prefix is not allowed. Using this prefix on MEM file hexadecimal values is flagged as a syntax error.

There must be at least one data value following an address, up to as many data values that belong to the previous address value. Following is an example of the most common MEM file format:

```
@0000 3A @0001 7B @0002 C4 @0003 56 @0004 02
@0005 6F @0006 89...
```

Data2MEM requires a less redundant format. An address specifier is used only once at the beginning of a contiguous block of data. The previous example is rewritten as:

```
@0000 3A 7B C4 56 02 6F 89...
```

The address for each successive data value is derived according to its distance from the previous address specifier. However, the derived addresses depends on whether the file is being used as an input or output. The differences between input and output memory files are described in Memory Files as Output and Memory Files as Input.

A MEM file can have as many contiguous data blocks as required. While the gap of address ranges between data blocks can be any size, no two data blocks can overlap an address range.

Data2MEM use the following memory type keywords:

- **RAMB16**
- **RAMB18**
- **RAMB32**
- **RAMB36**

For information on valid memory types by device and primitive, see Block RAM Configurations by Device Family.

Memory (MEM) Files As Output

Output Memory (MEM) files are used primarily for Verilog simulations with third-party memory models. The format adheres to the following industry standards:

- 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 not a true *address*. It is an *index offset* from the beginning of the larger array of where the data should begin. For example, the following MEM fragment indicates that data starts at the 655th hexadecimal location (given that indexes start at zero), within an array of 16-bit data values:

  `@654 24B7 6DF2 D897 1FE3 922A 5CAE 67F4...`

- If an address gap exists between two contiguous blocks of data, the data between the gaps still logically exists, but is undefined. For information on using the **OUTPUT** keyword to generate output MEM files, see BMM File Syntax.

Memory (MEM) Files as Input

Input Memory (MEM) files have format restrictions that do not conform to industry standards.

- White space between adjacent data values is ignored. Instead, all the values in contiguous blocks of data are treated as continuous streams of bits. Data2MEM breaks the bitstream up into data values according to the width to which the target block RAMs are configured. White space between adjacent data values is used solely for readability.

- An address specifier must reside within an address space range defined in a BMM file.

  Note The specifier is not specifically a CPU memory address. Instead, the specifier is any number that matches a BMM address space.

- Derived addresses for successive data values depend on the byte length of the value, despite the fact that address specifiers are not specifically CPU memory address. An 8-bit value increments the next derived address by one, a 16-bit value by two, 32-bit value by four, and so forth.

- If an address gap exists between two contiguous blocks of data, the address gap is assumed to be a 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.

Memory (MEM) Files Using Parity

When parity is used, Data2MEM assumes the upper (most significant) Bit Lane data bits are connected to the parity data bits of a block RAM. Since hexadecimal format only allows values to be defined in even 4 bit nibble values, hexadecimal digits must be added to the most significant end of a value to accommodate the additional parity bits. Data2MEM knows the data bus width of a block RAM, and because of the fixed 4 bit width of hexadecimal digits, Data2MEM discards any additional bits added by the hexadecimal 4 bit width restriction.

For example, an 18 bit data value, `0x23A24`, can be specified in hexadecimal only as a 20 bit value. In this example, the bottom most, least significant bits of the most significant nibble (bits 17 and 16) contain the value `0x2`. However, the top two significant bits of the nibble (bits 19 and 18) are unused. Because Data2MEM knows the data width of the data bus on the Block RAM is 18 bits wide, it discards the two data bits. Similarly, a 9 bit data value, `0x1D4`, would have the top three data bits discarded. This discarding process is also used with non-parity Block RAM data widths less than 4 bits wide, such as 1 and 2.

Bitstream (BIT) Files

A bitstream (BIT) file is a binary data file that contains a bit image to be downloaded to an FPGA device. Data2MEM can directly replace the block RAM data in BIT files without using any Xilinx® implementation tools. Consequently, Data2MEM both inputs and outputs BIT files. However, Data2MEM can only modify existing BIT files. A BIT file is initially generated by the Xilinx implementation tools.

Because BIT files are binary, you cannot edit a BIT file directly.

Data2MEM allows you to view BIT file content. For more information, see Using the Command Line.

Verilog Files

A Verilog file:

- Is a text file output by Data2MEM as a `.v` file
- Contains `defparam` records to initialize Block RAMs
- Is used primarily for pre- and post-synthesis simulation

Although you can edit a Verilog file directly, Xilinx® recommends that you do not do so since it is a generated file. If you do edit the file, Data2MEM allows the free-form use of both `//` and `/*...*/` commenting styles.

VHDL Files

A VHDL file:

- Is a text file output by Data2MEM as a `.vhd` file
- Contains `bit_vector` constants to initialize block RAMs. These constants can be used in generic maps to initialize an instantiated block RAM.
- Is used primarily for pre- and post-synthesis simulation

is . It

Although you can edit a VHDL file directly, Xilinx® recommends that you do not do so since it is a generated file. If you do edit the file, Data2MEM allows the free-form use of both `//` and `/*...*/` commenting styles.

User Constraints File (UCF) Files

A User Constraints File (UCF):

- Is a text file that is output by Data2MEM
- Contains INIT constraints to initialize Block RAMs

Although you can edit a UCF file directly, Xilinx® recommends that you do not do so since it is a generated file. If you do edit the file, Data2MEM allows the free-form use of both `//` and `/*...*/` commenting styles.

UCF files are supported for legacy workflows. Xilinx discourages their use for new designs or workflows.

Block RAM Memory Map (BMM) File Syntax

The chapter discusses the syntax used in Block RAM Memory Map (BMM) files. This chapter includes:

- Block RAM Memory Map (BMM) Features
- Address Map Definitions (Multiple Processor Support)
- Address Space Definitions
- Bus Block Definitions (Bus Accesses)
- Bit Lane Definitions (Memory Device Usage)
- Auto Block RAM Memory Map (BMM) Creation

Block RAM Memory Map (BMM) Features

A Block RAM Memory Map (BMM) file is designed for human readability. It is similar to high-level computer programming languages in using the following features:

- Block structures by keywords or directives

  BMM maintains similar structures in groups or blocks of data. BMM creates blocks to delineate address space, bus access groupings, and comments.

- Symbolic name usage

  BMM uses names and keywords to refer to groups or entities (improving readability), and uses names to refer to address space groupings and Block RAMs.

- In-file documentation

  BMM files allow comment blocks anywhere within the content of the file.

- Implied algorithms

  BMM allows you to specify data transposition in semi-graphical terms, while alleviating the need to specify the exact details of the address-to-block RAM algorithm. The software then infers the algorithm details for the desired mapping.

BMM observes the following conventions:

- Keywords are case-sensitive
- Keywords are uppercase
- Indenting is for clarity only. For a recommended style, see the Example BMM file in Block RAM Memory Map (BMM) Files.
- White space is ignored except where it delineates items or keywords.
- Line endings are ignored. You can have as many items as you want on a single line.
- Comments can one of two types:
  - `/*...*/`

    Brackets a comment block of characters, words, or lines. This type of comment can be nested.
  - `//`

    Everything to the end of the current line is treated as a comment.
- Numbers can be entered as decimal or hexadecimal. Hexadecimal numbers use the `0xXXX` notation form.

For more information on the modifications to Backus-Naur form syntax used in BMM, see BMM Modified Backus-Naur Form Syntax .

Address Map Definitions (Multiple Processor Support)

Data2MEM supports multiple processors using the following keywords:

- **ADDRESS_MAP**
- **END_add_MAP**

The syntax for the keywords is:

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

These keywords surround the **ADDRESS_SPACE** definitions that belong to the memory map for a single processor.

- **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.

Each processor has its own **ADDRESS_MAP** definition.

ADDRESS_SPACE names must be unique only within a single **ADDRESS_MAP**. Normally, instance names must be unique within a single **ADDRESS_MAP**. However, Data2MEM requires instance names to be unique within the entire BMM file.

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

- Any specific **ADDRESS_SPACE** is referred to by its **map_name** and **space_name** name separated by a period (or dot) character, for example, **cpu1.memory**.
- The address tag name can be shortened to just the **ADDRESS_MAP** name, which confines data translation to only those **ADDRESS_SPACES** within the named **ADDRESS_MAP**. This is used to send data to a specific processor without having to name each individual **ADDRESS_SPACE**.

For backward compatibility, **ADDRESS_SPACE** can still be defined outside 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

The outermost definition of an address space is composed of the following components:

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

The **ADDRESS_SPACE** and **END_add_SPACE** block keywords define a single contiguous address space. The mandatory name following the **ADDRESS_SPACE** keyword provides a symbolic name for the entire address space. Referring to the address space name is the same as referring to the entire contents of the address space, as illustrated in High Level Software and Hardware Tool Flows in Uses for Data2MEM.

A BMM file can contain multiple **ADDRESS_SPACE** definitions, even for the same address space, as long as each **ADDRESS_SPACE** name is unique.

Following the address space name is a keyword that defines from the type of memory device from which the **ADDRESS_SPACE** is constructed. The following device types are defined:

- **RAMB16**
- **RAMB18**
- **RAMB32**
- **RAMB36**
- **MEMORY**
- **COMBINED**

In Spartan®-3 devices and Virtex®-4 devices:

- The **RAMB16** keyword defines the memory as a 16-Kbit block RAM without parity included.
- The **RAMB18** keyword defines the memory space as an 18-Kbit block RAM using parity.

In Virtex-5 devices:

- The **RAMB32** keyword defines the block RAM memory size and style as a non-parity memory.
- The **RAMB36** keyword defines a 36 Kbit block RAM using parity memory.

You must use the correct keyword for the memory size and style selected.

The **MEMORY** keyword defines the memory device as generic memory. In this case, the size of the memory device is derived from the address range defined by the **ADDRESS_SPACE**.

For more information on the **COMBINED** keyword, see Auto BMM

Following the memory device type is the address range that the Address Block occupies by using the **[start_addr:end_addr]** pair.

The **end_addr** is shown following the **start_addr**, but the actual order is not mandated. For either order, Data2MEM assumes that the smaller of the two values is the **start_addr**, and the larger is the **end_addr**.

Bus Block Definitions (Bus Accesses)

Inside an **ADDRESS_SPACE** definition are a variable number of sub-block definitions called Bus Blocks.

```
BUS_BLOCK
   Bit_lane_definition
   Bit_lane_definition   .   .
END_BUS_BLOCK;
```

Each Bus Block contains block RAM Bit Lane definitions that are accessed by a parallel CPU bus access. In the *Example Block RAM Address Space Layout*, there are 4 Bus Block rows with each Bus Block containing 8 bit lines of 8 bits each.

The order in which the Bus Blocks are specified defines which part of the address space a Bus Block occupies. The lowest addressed Bus Block is defined first, and the highest addressed Bus Block is defined last. In the Example BMM file in Block RAM Memory Map (BMM) Files the first Bus Block occupies CPU addresses 0xFFFFC000 to 0xFFFFCFFF. This is the same as the first row of block RAMs in the Example Block RAM Address Space Layout figure shown in Data2MEM Design Considerations for Block RAM-Implemented Address Space. The second Bus Block occupies CPU addresses 0xFFFFD000 to 0xFFFFDFFF, which represents the second row of block RAM shown in the figure. This pattern repeats in ascending order until the last Bus Block.

The top-to-bottom order in which Bus Blocks are defined also controls the order in which Data2MEM fills those Bus Blocks with data.

Bit Lane Definitions (Memory Device Usage)

A Bit Lane definition determines which bits in a CPU bus access are assigned to particular block RAMs. Each definition takes the form of a block RAM instance name followed by the bit numbers the Bit Lane occupies. The instance name must be preceded by its hierarchical path as used in the system design. The syntax is as follows:

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

When parity is used, Data2MEM assumes that the upper (most significant) Bit Lane data bits are connected to the parity data bits of the block RAM. For example, for a Bit Lane that is defined as [17:0], data bits 15:0 are connected to the normal data bits of the block RAM, and bits 17 and 16 are connected to the parity bits of the block RAM.

Normally the bit numbers are given in the following order:

```
[MSB_bit_num:LSB_bit_num]
```

If the order is reversed to have the Least Significant Bit (LSB) first and the Most Significant Bit (MSB) second, Data2MEM bit-reverses the Bit Lane value so that [MSB_bit_num:LSB_bit_num] is represented before placing it into the block RAM.

As with Bus Blocks, the order in which Bit Lanes are defined is important. But in the case of Bit Lanes, the order infers which part of Bus Block CPU access a Bit Lane occupies. The first Bit Lane defined is inferred to be the most significant Bit Lane value, and the last defined is the least significant Bit Lane value. In the figure below, the most significant Bit Lane is BRAM7, and the least significant Bit Lane is BRAM0. As seen in *Example Block RAM Address Space Layout*, this corresponds with the order in which the Bit Lanes are defined.

When Data2MEM inputs data, it takes data from data input files in Bit Lane sized chunks, from the most significant value first to the least significant. For example, if the first 64 bits of input data are 0xB47DDE02826A8419 then the value 0xB4 is the first value to be set into a Block RAM.

Given the Bit Lane order, BRAM7 is set to 0xB4, BRAM6 to 0x7D, and so on until BRAM0 is set to 0x19. This process repeats for each successive Bus Block access BRAM set until the memory space is filled or until the input data is exhausted. The figure below expands the first Bus Block to illustrate this process.

Bit Lane Fill Order
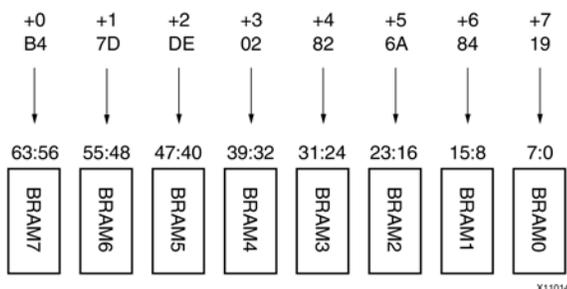
The Bit Lane definitions must match the hardware configuration. If the BMM is defined differently from the way the hardware actually works, the data retrieved from the memory components will be incorrect.

Bit Lane definitions also have some optional syntax, depending on what device type keyword is used in the Address Block definition.

When specifying **RAMB16** block RAM devices, the physical row and column location within the FPGA device can be indicated. Following are examples of the physical row and column location:

```
top/ram_cntlr/ram0 [7:0] LOC = X3Y5;
```

or

```
top/ram_cntlr/ram0 [7:0] PLACED = X3Y5;
```

Use the **LOC** keyword to place the corresponding block RAM to a specific location in the FPGA device. In this case the Block RAM is placed at row 3 and column 5 in the FPGA device. The **PLACED** keyword is inserted by the Xilinx® implementation tools when creating a *back-annotated* BMM file. For more information on back-annotated BMM files, see Using Integrated ISE® Implementation Tools. These definitions are inserted after the bus-bit values and the terminating semicolon.

An **OUTPUT** keyword can be inserted for outputting memory device MEM files. This takes the form of:

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

This specifier creates a memory (MEM) file with the data contents of the Bit Lane memory device. The output file name must end with the MEM file extension and can have a full or partial file path. The resulting MEM files can then be used as input to device memory models during a simulation run. As shown in the Example BMM file in Block RAM Memory Map (BMM) Files, MEM files are created for all the block RAMs in the second Bus Block.

In addition to using correct syntax for Bit Lane and Bus Block definitions, you must take into account the following limitations:

- While the examples in this document use only byte-wide data widths for clarity, the same principles apply to any data width for which a block RAM is configured.
- There cannot be any gaps or overlaps in Bit Lane numbering. All Bit Lanes in an Address Block must be the same number of bits wide.
- The Bit Lane widths are valid for the memory device specified by the device type keyword.
- The amount of byte storage occupied by the Bit Lane block RAMs in a Bus Block must equal the range of addresses inferred by the start and end addresses for a Bus Block.
- All Bus Blocks must be the same number of bytes in size.
- A block RAM instance name can be specified only once.
- A Bus Block must contain one or more valid Bit Lane definitions.
- An Address Block must contain one or more valid Bus Block definitions.

Data2MEM checks for all these conditions and transmits an error message if it detects a violation.

Combined Address Spaces

The BMM address space is synonymous with a memory controller. For every memory controller, a BMM address space is defined that describes the memory device elaboration for the memory controller.

The following coding example is a 4k address space for a single 32 bit bus memory controller with two block RAMs configured with 16 bit data buses.

```
ADDRESS_SPACE bram_block
RAMB16 [0x00000000:0x00000FFF]
   BUS_BLOCK
    bram0 [31:16];
    bram1 [15:0];
  END_BUS_BLOCK;
END_add_SPACE;
```

As long as the one-to-one (address space to memory controller) relationship can be maintained, the coding example is valid. However, current designs do not necessarily maintain a one-to-one relationship between address space to memory controller. Current memory controllers decode bus addresses only in powers of 2. If a memory size other than power of 2 is needed, multiple memory controllers with contiguous addressing must be used.

Current BMM files have two separate address space definitions for the 16k and 32k memory controllers. This instructs Data2MEM to treat the address spaces as two physically *separate* address spaces, even though the designer wants them to *logically* act as one.

If Data2MEM tries to translate data to the *logical* 48k address space that is larger than either *physical* 16k or 32k address spaces, an error occurs because data cannot span address spaces.

The previous resolution to this error was to hand-construct a BMM address space definition that contains all block RAM in both *physical* address spaces; however, this solution did not work if the two physical address spaces had block RAMs configured with different bus widths. However, to properly translate data, Data2MEM requires all block RAMs in an address space to be configured the same width.

To resolve the addressing of non-contiguous addresses, the BMM address space syntax instructs Data2MEM to combine *physical* address *ranges* into a single *logical* address space. This is accomplished by replacing the device type keyword in the address space header with the COMBINED keyword.

The following BMM code snippet describes a 12k address space for two 32 bit bus memory controllers:

- One memory controller with two block RAMs configured with 16 bit data buses.
- One memory controller with four block RAMs configured with 8 bit data buses.

A description of the distinctions between the first code example and this code example can be found after the code.

```
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_add_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_add_RANGE;
END_add_SPACE;
```

The distinctions between the two code examples are:

- The use of the COMBINED keyword for the memory type.
- The address values of the address space reflect the entire *logical* address space. Data2MEM can distinguished that the address space is constructed from several different *physical* address *ranges*.
- The use of the keyword block structure, ADDRESS_RANGE and END_add_RANGE. Each address range brackets the memory elaboration components just as the previous ADDRESS_SPACE definitions in the first example contained all of the BUS_BLOCKs and Bit Lanes.

The address range header contains the type of memory component from which the address range is constructed (in this case, **RAMB16**). When Data2MEM translates data that exceeds the first address range, translation continues automatically with the next address range.

Because each address range defines its own memory component type, each address range can use different memory types, such as block RAM, external memory, and Flash. A *logical* address space can therefore be a mix of *physical* memory types, which provides a greater range of flexibility in memory options.

Auto Block RAM Memory Map (BMM) Creation

Data2MEM automatically creates Block RAM Memory Map (BMM) files for single instantiations of block RAM. The Auto BMM feature allows the memory contents in the memory (MEM) files to change without rerunning Xilinx® tools during simulation. A simulation recompile is still necessary. Data2MEM is the only software tool needed to insert the new memory changes in the final BIT file.

Data2MEM automatically creates a BMM file based on the **INIT_fl** generic or parameter used on the instantiated block RAM. Data2MEM reads the **READ_WIDTH_A** block RAM value to determine the data width.

For data widths of 8 or greater, Data2MEM assumes that parity bits are in the MEM file.

The following VHDL and Verilog coding examples use the **INIT_fl** generic or parameter.

VHDL Coding INIT_fl Example -

```
ramb16_0 : RAMB16
 generic map (INIT_fl => "file.mem",
  :
  : )
 port map ( ... );
```

Verilog Coding INIT_fl Example -

```
RAMB16 #(.INIT_fl("file.mem")
        ...)
ramb16_0 ( <port mapping>);
```

Using the Command Line

This chapter describes command line functionality and includes:

- Checking Memory Map (BMM) File Syntax
- Translating or Converting Data Files
- Translating Data Files with Tag or Address Block Name Filtering
- Replacing Bitstream (BIT) File Block RAMs
- Displaying Bitstream (BIT) and ELF File Contents
- Ignoring Executable and Linkable Format (ELF) and Memory (MEM) Files Outside Address Blocks in Memory Map (BMM)
- Forcing Text Output Files for Address Spaces

Checking Block RAM Memory Map (BMM) File Syntax

Use the **-bm** option to check the syntax of a BMM file. Run the following command:

```
data2mem –bm my.bmm
```

Data2MEM parses the BMM file `my.bmm` and reports any errors or warnings. If Data2MEM reports no errors or warnings, the BMM file is correct.

Data2MEM checks only the BMM syntax. You must still ensure that the BMM file matches the logic design.

Translating or Converting Data Files

Use the **-bd** and **-o** options with the **-bm** option to transform Executable and Linkable Format (ELF) or memory (MEM) data files into other formats.

Data files are converted into block RAM initialization files for Verilog and VHDL, or User Constraints File (UCF) block RAM initialization records. Run the following command to convert to all three formats:

```
data2mem –bm my.bmm –bd code.elf -o uvh output
```

This command generates the following files:

- `output.v`
- `output.vhd`
- `output.ucf`

While only one data file is shown here, you can specify as many `–bd` data file pairs as you need. These files can then be incorporated directly into the design source file set, or used in simulation environments.

Another conversion is a variation of dumping the contents of ELF files. Using dump in this way effectively converts ELF files to MEM files. Run the following command:

```
data2mem –bd code.elf –d -o m code.mem
```

The `code.mem` file contains a text version of the binary ELF file. This is useful for making patches to ELF files when the source file is no longer available.

ELF or MEM data files can be translated into device initialization MEM files. The linear data in the input data files is converted to an initialization MEM file for the device that occupies a Bit Lane. This is true for both block RAM and external memory devices. Run the following command:

```
data2mem -bm my.bmm -bd code.elf -o m output
```

A Bit Lane appears as:

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

The resulting `ram0.mem` MEM file contains the initialization data for the `top/ram_cnlr/ram0` device only. This functionality is used primarily for simulation environments with external memory devices in the design.

The `output` file name is required, but is ignored. Instead, the output file name is controlled by the **OUTPUT** directive in the Bit Lane definition.

Translating Data Files with Tag or Address Block Name Filtering

Use Tag or Address Block name filtering to further control data file translation. Listing a set of Address Block names with each **-bd** option confines data translation to that set only. A **-bd** option can be modified as:

```
-bd code.elf tag mem1 mem2
```

Data translation takes place only for the Address Blocks **mem1** and **mem2**, even if data in `code.elf` matches another Address Block. This allows you to direct different data contents to Address Blocks that could have the same address range.

This method also allows you to restrict data translation to a portion of the design, leaving the rest of the design unaffected.

Tag name filtering implicitly invokes the **-i** option to turn off address space mismatch errors.

Replacing Bitstream (BIT) File Block RAMs

Data2MEM lets you insert new block RAM data into a Bitstream (BIT) file without rerunning the Xilinx® implementation tools. Together with a new ELF and BMM file, Data2MEM updates the block RAM initialization in a BIT file image and outputs a new BIT file. You can also use tag filtering.

Run the following command to create a new BIT file called `new.bit`, in which the appropriate block RAM contents are replaced with the contents of the `code.elf` file.

```
data2mem -bm my.bmm -bd code.elf -bt my.bit -o b new.bit
```

The BMM file *must* have LOC or PLACED constraints for each block RAM. These constraints can be added manually, but they are most often obtained as an annotated BMM file from Bitgen. For more information, see Using Integrated ISE® Design Suite Implementation Tools.

This method creates a new BIT file that improves speed from 100 to 1000 times over rerunning the implementation tools. This method is meant primarily as a way to include new CPU software code into a design when the logic portion of the design is not changing. You do not need to use the Xilinx implementation tools in order to add code.

Displaying Bitstream (BIT) and Executable and Linkable Format (ELF) File Contents

Data2MEM provides the ability to examine, or *dump*, the contents of Bitstream (BIT) and Executable and Linkable Format (ELF) files. The dump content is a text hexadecimal format representation of the input file and is printed to the console.

The **-d** option has two optional parameters that change the information displayed for the data files:

- The **e** parameter displays additional information about each section
- The **r** parameter displays redundant ELF header information

Run the following command to display ELF files:

```
data2mem -bd code.elf -d
```

ELF files contain more data (such as symbols and debug information) than those used by Data2MEM for data translation. Data2MEM considers only those sections labeled *Program header record* for data translation.

Run the following command to display Bit dumps:

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

Each bitstream command is decoded and displayed. Each bit field is described in those commands that contain bit field flags. Commands that contain non-block RAM data chunks display as plain hexadecimal dumps. Because block RAM data is encoded within bitstreams, Data2MEM displays block RAM data as decoded hexadecimal dumps.

These dumps are used primarily for debugging purposes. However, they can also be useful for comparing binary ELF and BIT files.

Ignoring Executable and Linkable Format (ELF) and Memory (MEM) Files Outside Address Blocks in Block RAM Memory Map (BMM) Files

The **-i** option instructs Data2MEM to ignore any data in an Executable and Linkable Format (ELF) or Memory (MEM) file that is outside any Address Block within the Block RAM Memory Map (BMM) file. This allows data files to be used that have more data in them than the BMM file recognizes. For example, Data2MEM can use a master design code file as an ELF data file, even though only a small portion of that file is data destined to be ELF code for block RAM memories.

Forcing Text Output Files for Address Spaces

The **-u** option forces Data2MEM to produce text output files for all Address Spaces, even if no data has been transformed into an Address Space. Depending on the file type, an output file is either empty, or contains an initialization of all zeroes. If the **-u** option is not used, only Address Spaces that receive transformed data are output.

Chapter 5

Using Integrated Implementation Tools

This chapter describes how Data2MEM functionality integrates with the Xilinx® implementation tool flow. This chapter includes:

- Using NGDBuild
- Using MAP and PAR
- Using Bitgen
- Using NetGen
- Using FPGA Editor
- Using iMPACT
- Integrated Implementation Tool Restrictions

The flow allows hardware designers to associate block RAM with a block RAM Memory Map (BMM) file directly from within the Xilinx implementation tools.

Access to Data2MEM functionality is obtained using a subset of Data2MEM options in Native Generic Database (NGD) NGDBuild, Bitgen, NetGen, and FPGA Editor.

The following figure illustrates the software flow and the file dependencies.

Software Flow and File Dependencies

CPU Software Tool Flow

FPGA Tool Flow

X11016
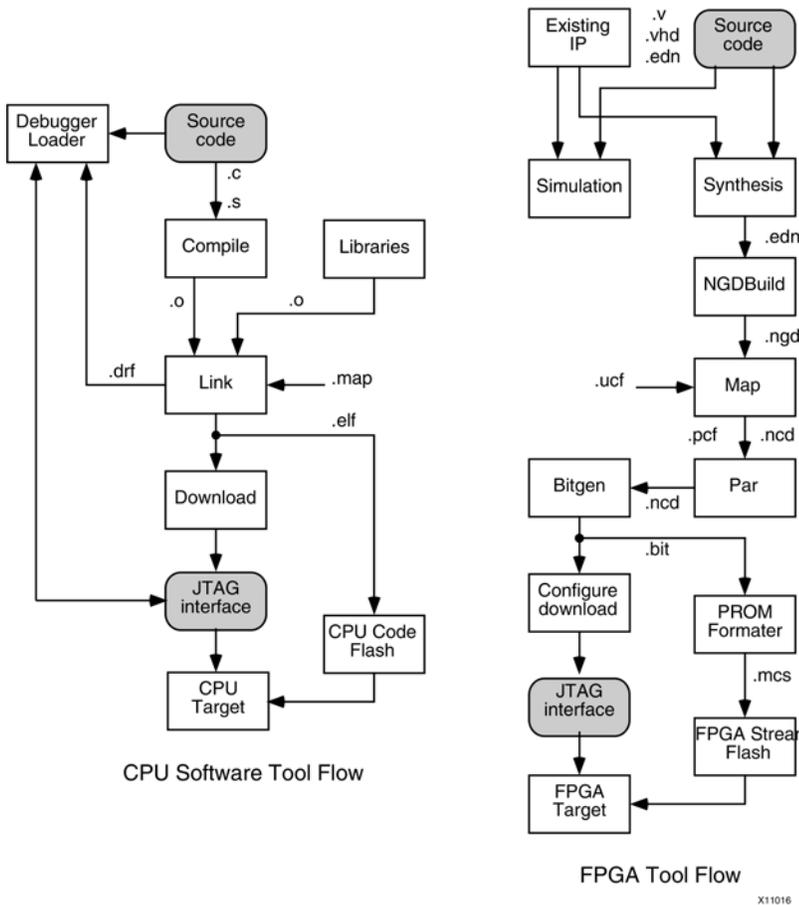
Note NGDBuild is a command that converts all input design netlists and then writes the results into a single merged file. The NetGen command prepares netlists for simulation.

Using NGDBuild

The **-bm** option allows you to specify the name and path of the BMM file.

```
Option: -bm
Syntax: -bm filename[.bmm]
```

If you add the BMM file to an ISE® Design Suite project, ISE Design Suite tracks changes to the BMM file and re-implements the design when necessary.

NGDBuild creates the **BMM_fl** property in the Native Generic Database (NGD) file to signal that a BMM file design is being used. After the BMM file is syntax-checked, NGDBuild validates that the block RAMs referenced in the BMM file actually exist. You can also syntax check the BMM file by running the command line version of Data2MEM. Any block RAM placement constraints in the BMM file are applied to the corresponding block RAM.

ISE Design Suite supports the **-bm** option. For information on how to set the **-bm** option so that it is invoked in NGDBuild, see *Translate Properties* in ISE Design Suite Help.

Using MAP and PAR

There are no command line or functionality changes to MAP or Place and Route (PAR).

You must correctly connect the block RAM components. MAP trims incorrectly connected block RAM components.

To determine if any block RAM components were removed, see *Section 5 - Removed Logic* in the MAP report.

Using Bitgen

The **-bd** switch specifies the path and file name of the Executable and Linkable Format (ELF) file used to populate the block RAMs specified in the BMM file.

```
Option: -bd
Syntax: -bd filename[.elf|.mem] [<tag TagName...>]
```

The address information contained in the ELF file allows Data2MEM to determine which **ADDRESS_SPACE** to place the data.

Bitgen passes the **-bd** switch with the `<filename>` and any tag information to Data2MEM. Data2MEM processes the BMM file specified during the NGDBuild phase. The ELF file is used to internally create the block RAM initialization strings for each BMM-defined block RAM. The initialization strings are then used to update the Native Generic Database (NGD) file before the BIT (bitstream) file is created.

Placement information of each block RAM is provided by the NCD file. Any block RAM placement constraints that appear in the BMM file are already reflected in the NCD information. All other block RAMs are assigned placement constraints by previous tool steps. These placement constraints are passed to Data2MEM by the `<BMMfilename>_bd.bmm` file, a back-annotated BMM file.

If Bitgen is invoked with a NCD file that contains a `BMM_fl` property, but a **-bd** option is *not* given, a back-annotated BMM file is still produced. The corresponding block RAMs will have zeroed content.

In addition to the original BMM file contents, this file contains the placement information for all the block RAMs defined by the BMM file. The back-annotated BMM file and the resulting BIT file can then be used to perform direct BIT file replacement with the command line version of Data2MEM.

ISE® Design Suite supports the **-bd** switch. For more information, see
http://www.xilinx.com/products/design_resources/design_tool/index.dita.

Using NetGen

ISE® Design Suite supports the **-bd** and **-bx** options. For information on setting these options for NetGen, see
http://www.xilinx.com/products/design_resources/design_tool/index.dita .

Using the -bd Option

```
-bd <elf_flname>[.elf | .mem]
```

The **-bd** option specifies the path and name of the ELF file used to populate the block RAMs specified in the BMM file.

The address information in the ELF file allows Data2MEM to determine the **ADDRESS_SPACE** in which to place the data.

1. NetGen passes the **-bd** option with the `<elf_flname>` to Data2MEM.

2. Data2MEM processes the BMM file specified during NGDBuild.

3. The ELF file is used to create the block RAM initialization strings for each of the constrained block RAMs.

4. The initialization strings are used to update the NCD file before the BIT file is created.

5. The NCD file passes the block RAM placement information to Data2MEM.

6. The block RAM placement information is used to create the `<bramfilename>_bd.bmm` file.

7. The `<bramfilename>_bd.bmm` file contains the placement information for all block RAM, constrained or unconstrained, in order to enable the command line version of Data2MEM.

Using the -bx Option

```
-bx [filepath]
```

The **-bx** option specifies the file path to output individual memory device MEM files for performing Hardware Description Language (HDL) simulations from the contents supplied in the **-bd** options. The **-bd** option must be supplied when using **-bx** option.

1. NetGen passes the **-bx** option with the optional `filepath` to Data2MEM.

2. Data2MEM outputs the individual MEM files to the supplied file path. If no `filepath` is supplied, Data2MEM default to the current working directory. If `filepath` is supplied, that file path must already exist. The file path is not generated automatically.

3. In the resulting netlist output, each block RAM instance is annotated with an **INIT_fl** parameter to indicate the MEM file with which to initialize the block RAM. During subsequent simulations, the memory file in the **INIT_fl** parameter is used to initialize the contents of the block RAM.

4. This functionality is currently only available to Virtex®-4 devices and Virtex-5 devices.

Updated MEM files can be created by running Data2MEM alone using the **-bx** option. This avoids the necessity of rerunning NetGen to generate updated MEM files.

Using FPGA Editor

The **-bd** option specifies the path and file name of the ELF file used to populate the block RAMs specified in the BMM file.

```
Option: -bd
Syntax: -bd <elf_flname>[.elf | .mem]
```

The address information contained in the ELF file allows Data2MEM to determine the **ADDRESS_SPACE** in which to place the data.

FPGA Editor marks the block RAMs specified in the BMM file as read-only. Any changes made in FPGA Editor to the contents of the block RAM mapped in the BMM file are not retained, even if you write out the NCD. To change the contents of the block RAMs permanently, you must change the ELF file.

ISE® Design Suite supports the **-bd** switch. For more information, see http://www.xilinx.com/products/design_resources/design_tool/index.dita.

Using iMPACT

The Data2MEM translation process occurs on-the-fly. As iMPACT downloads, a bitstream (BIT) file is configured into an FPGA device. The sequence is:

1. Open the configuration dialog box in iMPACT.

2. Choose the block RAM Memory Map (BMM) files

3. Choose the Executable and Linkable Format (ELF) files

4. Choose tag names to associate with the ELF files, This confines ELF file translation to the chosen tag name **ADDRESS_SPACE** keywords only.

5. Enter a boot address for each destination processor.

iMPACT Process Flow

iMPACT reads the BIT file and passes the following to Data2MEM:

1. A memory image of the BIT file.

2. The BMM file.

3. The ELF files with any tag names.

- Data2MEM translates any ELF data that matches **ADDRESS_SPACE** keywords in the BMM and replaces any block RAM contents in the BIT file memory image. The memory image is handed back to iMPACT.

- iMPACT configures the updated BIT file memory into the FPGA device and halts any processors.

- iMPACT then requests any external memory data from Data2MEM. Data2MEM passes back any external memory data along with its starting address and size, with the JTAG ID of the destination processor (as defined in the multi-process support description above). iMPACT sends the data to the processor via JTAG with *instruction-stuffing* commands.

- The halted processor performs bus cycles to store the data in the destination external memory. The process is repeated until all external memory data for all ADDRESS_MAP structures defined in the BMM file have been initialized.

- iMPACT requests the boot address from Data2MEM for each ADDRESS_MAP structure defined in the BMM file. The boot address either comes from the last ELF file to be translated to a ADDRESS_MAP structure, or it can be overridden as part of the initial configuration dialog box options.

- iMPACT sets and the boot address and restarts each processor. The processor then starts execution at its boot address.

This process is used during the development phase. It allows rapid turn around for downloading new test software. The same process can be used to generate an `Serial Vector Format (SVF)` file by instructing iMPACT to direct the configuration stream into a file instead of an FPGA device. The iMPACT tool can then translate the SVF file into an ACE file to use with System ACE™ technology. This puts the configure stream into a shippable form, with complete configure, block RAM, and external memory initialization.

The processors must be the first devices in the JTAG chain, and the JTAG IDs must match the `processor_ids` in the BMM file. If an **ADDRESS_MAP** structure for a processor does not exist in the BMM file, that processor will not be initialized.

Integrated Implementation Tool Restrictions

The following restrictions apply to the use of Data2MEM integrated implementation tools:

- For tools, XDL does not call Data2MEM to update the block RAM initialization strings. This results in different values from those seen in FPGA Editor, BitGen, and NetGen.

- The block RAMs specified in the BMM file can be trimmed during MAP if connected incorrectly. This may result in an error when Data2MEM is run.

- Translating CPU addresses to physical block RAM addresses must be done as part of the hardware design.

Command Line Syntax and Options

This chapter discusses command line syntax and options and includes:

- Command Line Syntax
- Command Line Options
- BMM Modified Backus-Naur Form Syntax

Command Line Syntax

```
<-bm FILENAME [.bmm]> |<<[-bm FILENAME [.bmm]]>
<-bd FILENAME [<.elf>|<.mem>] [<[<boot [ADDRESS]>] tag TagName <TagName>...>]>
  <-o <u|v|h|m> FILENAME [.ucf|.v|.vhd|.mem]>
  <-p PARTNAME>-i>> |
<<-bd FILENAME [.elf]> -d [e|r]>[<-o m FILENAME [.mem]>]>> |
<<-bm FILENAME [.bmm]>
<-bd FILENAME [<.elf>|<.mem>] [<[<boot [ADDRESS]>] tag TagName <TagName>...>]>
<-bt FILENAME [.bit]> <-o b FILENAME [.bit]>> |
<<-bm FILENAME [.bmm]>
<-bt FILENAME [.bit]> -d>> |<-bx [FILEPATH]> |
<-mf <p PNAME PTYPE PID
  <a SNAME MTYPE ASTART BWIDTH
  <s BSIZE DWIDTH IBASE>...>...>...>> |
<<-pp FILENAME [.bmm]>
<-o p FILENAME [.bmm]>> |
<-f FILENAME [.opt]> |
<-w [on|off] > |<-q [s|e|w|i]> |
<-intstyle silent|ise|xflow> |
<-log [FILENAME [.dmr]]> |
 <-u> |
 <-h [ <option [< option>...]> | support ]>
```

Command Line Options

Command	Description
-bm *filename*	Name of the input Block RAM Memory Map (BMM) file. If the file extension is missing, a BMM file extension is assumed. If this option is not specified, the Executable and Linkable Format (ELF) or memory (MEM) root file name with a `.bmm` extension is assumed. If only this option is given, the BMM file is syntax-checked only and any errors are reported. As many **-bm** options can be used as are required.

Command	Description
-bd *filename*	Name of the input ELF or MEM files. If the file extension is missing, the `.elf` extension is assumed. The `.mem` extension *must* be supplied to indicate a MEM file.
	If **TagNames** are given, only the address space of the same names within the BMM file are used for translation. All other input file data outside of the **TagName** address spaces are ignored. If no further options are specified, **-o u** *filename* functionality is assumed. As many **-bd** options can be used as are required.
	TagName has two forms:
	Names on the processor memory map, (`ADDRESS_MAP/END_add_MAP`) structure. This allows a single name to refer to the entire group of encapsulated `ADDRESS_SPACE`s. Specifying only the **TagName** of a processor confines the data translation to the `ADDRESS_SPACE` of that processor. To refer to any specific `ADDRESS_SPACE` within a named processor **TagName** group, use the **TagName** of the processor followed by a '.' (dot or period) character, then the **TagName** of the `ADDRESS_SPACE`. For example:
	`cpu1.memory`
	For backwards compatibility, `ADDRESS_SPACE`s that are defined outside of any `ADDRESS_MAP/END_add_MAP` structure are encapsulated inside an implied null processor name. These `ADDRESS_SPACE`s are, therefore, still refered to with just their `ADDRESS_SPACE` name as its TagName.
	TagName keywords are:
	`tag`, which separates the address space names from the data file name.
	`boot`, which identifies the data file as containing the boot address for a processor, and must preceded the `tag` keyword. If the optional `ADDRESS` value follows the `boot` keyword, then the `ADDRESS` value overrides the boot address of the data file. Only one `boot` keyword can be used for each processor TagName group. If no `boot` keyword is used for a processor TagName group, then the last `-bd` data file encountered for that group is used for the boot address of the processor.
-bx *filepath*	File path to output individual memory device MEM files for performing Hardware Description Language (HDL) simulations. If an **OUTPUT** keyword exists on a Bit Lane, the supplied MEM file name is used for output. Otherwise, the output MEM file name will be a combination of the Address Space name and a numeric value appended to the end of the Address Space name. If **TagName** are given, only the address spaces of the same names within the BMM file will be used for translation. All other input file data outside of the **TagName**TagName address spaces will be ignored. As many **-bx** options can be used as are required.
	TagName has two forms:
	• Names on the processor memory map, (`ADDRESS_MAP/END_add_MAP`) structure. This allows a single name to refer to the enitre group of encapsulated `ADDRESS_SPACE`s. Specifying only the **TagName** of a processor confines the data translation to the `ADDRESS_SPACE` of that processor. To refer to any

Command	Description
	specific `ADDRESS_SPACE` within a named processor **TagName** group, use the **TagName** of the processor followed by a '.' (dot or period) character, then the **TagName** of the `ADDRESS_SPACE`. For example: `cpu1.memory` • For backwards compatibility, `ADDRESS_SPACEs` that are defined outside of any `ADDRESS_MAP/END_add_MAP` structure are encapsulated inside an implied null processor name. These `ADDRESS_SPACEs` are, therefore, still refered to with just their `ADDRESS_SPACE` name as its `TagName`. **TagName** keywords are: • **tag**, which separates the address space names from the data file name. • **boot**, which identifies the data file as containing the boot address for a processor, and must preceded the **tag** keyword. If the optional **ADDRESS** value follows the **boot** keyword, then the `ADDRESS` value overrides the boot address of the data file. Only one **boot** keyword can be used for each processor **TagName** group. If no **boot** keyword is used for a processor **TagName** group, then the last **-bd** data file encountered for that group is used for the boot address of the processor.
-bt *filename*	Name of the input bitstream (BIT) file. If the file extension is missing, a `.bit` file extension is assumed. If the **-o** option is not specified, the output BIT file name has the same root file name as the input BIT file, with an **_rp** appended to the end. A .bit file extension is assumed. Otherwise, the output BIT file name is as specified in the **-o** option. Also, the device type is automatically set from the BIT file header, and the **-p** option has no effect.
-ou|v|h|m|b|p|d *filename*	The name of the output file(s). The string preceding the `filename` indicates which file formats are to be output. No spaces can separate the `filetype` characters but they can appear in any order. As many *filetype* characters as are required can be used. The *filetype* characters are defined as follows: • **u** = UCF file format, a .ucf file extension. • **v** = Verilog file format, a .v file extension. • **h** = VHDL file format, a .vhd file extension. • **b** = BIT file format, a .bit file extension. • **p**= Preprocessed BMM information, a .bmm file extension. • **d**= Dump information text file, a .dmp file extension. The `filename` applies to all specified output file types. If the file extension is missing, the appropriate file extension is added to specified output file types. If the file extension is specified, the appropriate file extension is added to the remaining file formats. An output file contains data from all translated input data files. **Note** NOTE: the 'm' file type is no longer used for outputting MEM files for individual memory devices. See the **-bx** option for outputting these MEM files.

Command	Description
-u	Update **–o** text output files for all address spaces, even if no data has been transformed into an address space. Depending on file type, an output file is either empty or contains all zeroes. If this option is not used, only address spaces that receive transformed data are output.
-mf *<BMM info items>*	Create a BMM definition. The following items define an Address Space within the BMM file. All items must be given, and must appear in the order indicated. As many groups of items can be given as required to define all Address Spaces within the BMM file. As many separate **–mf** options definitions as needed can used.
	These definitions can substitute for a **–bm** file option, or can be used to generate a BMM file. Use the **–o p** `filename` option to output a generated BMM file. The syntax is contained in four groups, and can be combined into a single **–mf** option.
-mf *<MNAME MSIZE <MWIDTH [MWIDTH...]>*	The User Memory Device Definition group items mean: • m = The following three items define an User Memory definition. This allows non-BRAM memories to be defined as large, and with available configuration bits widths as needed. A User Memory Device definition must be defined before its use, either in the same **–mf** option, or in a previous, separate **–mf** option. • **MNAME**= Alphanumeric name of a User Defined Memory Device. • **MSIZE** = Hex byte size of the User Memory Device. For example, `0x1FFFF`. • **MWIDTH** = Numeric bit widths the User Memory Device can be configured to as many bit widths can be specified as needed. root instance name. Values start at zero.

-mf<p PNAME PTYPE PID <a ANAME ['x' | 'b'] ASTART BWIDTH <s MTYPE BSIZE DWIDTH IBASE>	The Address Space Definition group items mean:
	
	 • **p** = The following three items define an Address Map definition. As many Address Map definitions are repeated back-to-back. An Address Map definition must have at least one Address Space definition.
	
	   – **PNAME** = Alphanumeric name of the Processor Map.
	
	   – **PTYPE** = Alphanumeric name of the processor type in the Address Map. Legal processor types are `PPC405`, `PPC440` and `MB`
	
	   – **PID** = Numeric ID of the Address Map.
	
	 • **a** = The following three items define an Address Space definition for the previous Address Map definition. As many Address Space definitions as needed are repeated back-to-back. An Address Space definition must have at least one Address Range definition.
	
	   – **ANAME** = Alphanumeric name of the Address Space.
	
	   – **'x'|'b'** = Addressing style for the Address Space definition. A 'b' specifies byte addressing. Each LSB increment will advance the addressing by one byte. A 'x' specifies index addressing. Each LSB increment will advance the addressing in `BWIDTH` bit sized values. This item is optional, and if missing, byte addressing is assumed.

Command	Description

	– **ASTART** = Hex address the Address Space starts from. For example, `0xFFF00000`.
	
	    – **BWIDTH** = Numeric bit width of the bus access for the Address Space.
	
	    – **MTYPE** = The memory type the Address Range is construct of. Legal memory types are `''RAMB16'`, `'RAMB18'`, `'RAMB32'`, `'RAMB36'`, and any user-defined memory device.
	
	    – **BSIZE** = Hex byte size of the Address Range. For example, `0x1FFFF`.
	
	    – **DWIDTH** = Numeric bit width of each Bit Lane within an Address Range.
	
	    – **IBASE** = Base alpha numeric hierarchy/part instance name assigned to each bitLane device. To make each instance names unique, an increasing numeric value is appended to the right end.

-mf <t*TNAME* [*'x'*|*'b'*]*ASIZE BWIDTH*<s *MTYPE BSIZE DWIDTH IBASE*>	The Address Template Definition group items mean: • **t** = The following three items define an Address Template definition. This defines the static aspects of an address space. Then one template can be used to instance multiple address spaces. • **TNAME** = Alpha numeric name of the Address Template. • **'x'|'b'** = Addressing style for the Address Template definition. • **b** specifies byte addressing. Each LSB increment will advance the addressing by one byte. • **x** specifies index addressing. Each LSB increment will advance the addressing in **BWIDTH** bit sized values. This item is optional, and if missing, byte addressing is assumed. • **s** = The following four items define an Address Range definition, for the previous Address Space definition. As many Address Range definitions as needed are repeated back-to-back. • **SIZE** = The hexadecimal byte size of the Address Template. For example, `0x1FFFF` • **BWIDTH** = Numeric bit width of the bus access for the Address Template. • **MTYPE**= The memory type the Address Range is construct of. Legal memory types are: `RAMB16`, `RAMB18`, `RAMB32`, `RAMB36`, and any User Defined Memory Device. • **BSIZE**= Hex byte size of the Address Range. For example, `0x1FFFF`. • **DWIDTH** = Numeric bit width of each Bit Lane within an Address Range. • **IBASE**= Base alpha-numeric hierarchy/part instance name assigned to each Bit Lane device. To make each instance names unique, an increasing numeric value is appended to the right end of the root instance name. Values start at zero. In addition, a base instance path can begin with a `'*/'` syntax. When the template is

Command	Description
	expanded to an `instance`, `instance IROOT` and `IBASE` are joined together to form a complete instance path.
-mfI TNAME INAME ASTART IROOT	The Address Instance Definition group items are: • `i` = The following four items define an Address Instance definition. This defines the dynamic aspects of an address space. Then the instance items can be applied to an address template, to create a new address instance. • `TNAME`= Alphanumeric name of the Address Template. • `INAME` = Alphanumeric name of the Address Instance. • `ASTART` = Hex address the Address Space starts from. For example, `0xFFF00000`. `IROOT` = Root alphanumeric hierarchy/part instance name applied to each Bit Lane device. See the description for IBASE.
-pp *filename*	Name of the input preprocess file. If the file extension is missing, a **bmm** file extension is assumed. If a **–o p***filename* option is specified the preprocessed output is sent to that file. If the file extension is not specified, a .bmm extension is assumed. If a **–o p***filename* option is not specified, the preprocessed output is sent to the console. Input files do not have to be a BMM file; any text-based file can be used as input.
-p `partname`	Name of the target Virtex®-4 or Virtex-5 part. If this is unspecified, an `xcv50` part is assumed. Use the **–h** option to obtain the full supported part name list.
-d `e|r`	Dump the contents of the input ELF or BIT file as formatted text records. BIT file dumps display the BIT file commands, and the contents of each block RAM. When dumping ELF files, two optional modifier characters may follow the **–d** option. No spaces can separate the modifier characters, but can appear in any order. As many, or as few modifier characters as desired can be used at a given time. These modifiers are defined as follows: • `e` = EXTENDED mode. Display additional information for each ELF section. • `r` = RAW mode. This includes some redundant ELF information.
-i	Ignore ELF or MEM data that is outside the address space defined in the BMM file. Otherwise, an error is generated.
-f `filename`	Name of an option file. If the file extension is missing, an `.opt` file extension is assumed. These options are identical to the command line options but are contained in a text file instead. A option and its items must appear on the same text line. However, as many switches can appear on the same text line as desired. This option can be used only once, and a .opt file cannot contain a -f option.

Command	Description
-g e|w|i	Disable the output of Data2MEM messages. The string following the option indicates which messages types are disabled. No spaces can separate the message type characters, but the characters can appear in any order. As many, or as few message type characters as desired can be used at a given time. The message type string is optional. Leaving the message type blank is equivalent to using –q wi. The message type characters are defined as follows: • **e** = Disable ERROR messages. • **w** = Disable WARNING messages. • **i** = Disable INFO messages.
-h-h	Print help text, plus supported part name list.

BMM Modified Backus-Naur Form Syntax

```
Address_block_keyword   ::=   "ADDRESS_SPACE";
End_add_block_keyword ::=   "END_add_SPACE";
Bus_block_keyword   ::=   "BUS_BLOCK";
End_bus_block_keyword  ::=   "END_BUS_BLOCK";
LOC_location_keyword          ::=   "LOC";

PLACED_location_keyword          ::=   "PLACED";

MEM_output_keyword          ::=   "OUTPUT";

BRAM_location_keyword          ::=  LOC_location_keyword       |
          PLACED_location_keyword        ;

Memory_type_keyword         ::=
          "RAMB16" |"RAMB18" |"RAMB32"
          "MEMORY"|"COMBINED";

Number_range          ::=   "[" NUM ":" NUM "]";

Name_path         ::=   IDENT ( "/" IDENT )*;

BRAM_instance_name          ::=   Name_path;

MEM_output_spec          ::=  MEM_output_keyword       "=" Name_path [ ".mem" ];

BRAM_location_spec          ::=   BRAM_location_keyword       "="
          ( "R" NUM "C" NUM ) | ( "X" NUM "Y" NUM );

Bit_lane_def          ::=   BRAM_instance_name Number_range
          [ BRAM_location_spec | MEM_output_spec ] ";" ;

Bus_block_def          ::=   Bus_block_keyword
          ( Bit_lane_def )+
          End_bus_block_keyword ";" ;

Address_block_def          ::=   Address_block_keyword IDENT Memory_type_keyword Number_range
          ( Bus_block_def )+
          End_add_block_keyword ";" ;
```