Data2MEM User Guide

UG437 (v2.0)

XILINX®

The following table shows the revision history for this document.

	Version	Revision
04/12/07	1.0	Initial Xilinx® release.
12/04/07	2.0	Second release.

Table of Contents

Chapter 1: Introduction

Features 1
Overview 2
Uses for Data2MEM 2
CPU Software and FPGA Design Tool Flow 4
Data2MEM Operational Overview 5
Block RAM Configurations by Device Family 8

Chapter 2: Input and Output Files

Block RAM Memory Map (BMM) Files 11
Executable and Linkable Format (ELF) Files 12
Debugging Information Format DWARF (DRF) Files 12
Memory (MEM) Files 12
Memory Files as Output 13
Memory Files as Input 13
Memory Files Using Parity 14
Bitstream (BIT) Files 14
Verilog (V) Files 14
VHDL (VHD) Files 14
User Constraints Files (UCF) 15

Chapter 3: Block RAM Memory Map (BMM) File Syntax

Features 17
ADDRESS_MAP Definitions (Multiple Processor Support) 18
ADDRESS_SPACE Definitions (a Logical Address Space) 19
BUS_BLOCK Definitions (Bus Accesses) 20
Bit Lane Definitions (Memory Device Usage) 20
Combined Address Spaces 22

Chapter 4: Command Line Tool Usage

Block RAM Memory Map (BMM) File Syntax Checking 25
Data File Translation or Conversion 26
Data File Translation with Tag Name Filtering 26
Bitstream (BIT) File Block RAM Replacement 27
Examining BIT and ELF File Contents 27
Miscellaneous Functionality 28

Chapter 5: Using the Integrated Implementation Tools

Using NGDBuild 31

Using MAP and PAR 31
Using Bitgen 32
Using Netgen 33
Using FPGA Editor 34
Restrictions 34

Chapter 6: Integrated iMPACT Tool Usage

Overview 35
iMPACT Tool Process Flow 35

Chapter 7: Command Line Option Reference

Overview 37
Command Line Options and Descriptions 38

Appendix A: Example BMM file

Appendix B: BMM Modified Backus-Naur Form Syntax

Chapter 1

Introduction

This document describes how the Data2MEM software tool automates and simplifies setting the contents of Block RAM cells on Virtex™ devices, how this is used with the 32-bit CPU on the single-chip Virtex-II Pro devices, and how it is available on other Xilinx® FPGA family products.

This introduction contains the following:

- "Features"
- "Overview" which contains the following sections:
  - Uses for Data2MEM
  - CPU Software and FPGA Design Tool Flow
  - Data2MEM Operational Overview
  - Block RAM Configurations by Device Family

Features

The features of the Data2MEM are:

- Compatibility with the following families: Virtex™-II Pro/-II/-E/-4/-5 and Spartan™-II (-3/-E/-3E/-3A /-3AN/3aDSP)
- 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 (byte) 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 their natural file format.
- Reads MEM format (MEM) text files as input for Block RAM contents. The text format can be either hand or machine generated.
- Optionally produces formatted text dumps of the contents of Bit (BIT), ELF, and DRF files
- Produces Verilog and Very High Definition Language (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 any other Xilinx® 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 Unix) and uses them interchangeably.
- Allows the free-form use of `//` and `/*...*/` commenting syntax in text input files.

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 Field Programmable Gate Array (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. Additionally, Data2MEM provides a simplified means for initializing Block RAMs for non-CPU designs.

Data2MEM automates a complicated process to a simplified technique, and also accomplishes the following:

- Affects existing tool flows as little as possible, 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:

- Windows® 2000, with SP2 or higher, Windows XP, and Windows Vista
- Linux® OS

Uses for Data2MEM

You can use the Data2MEM tool for the following distinct processes:

1. In software design, as a command line tool for generating updated BIT files. Refer to Chapter 4, "Command Line Tool Usage" for additional information.
2. In hardware design, to integrate Data2MEM with the Xilinx implementation tools. Refer to Chapter 5, "Using the Integrated Implementation Tools" for additional information.
3. As a command line tool to generate behavioral simulation files.

Figure 1-1, page 3 is a high-level representation of how the two tool flows operate within discrete-chip CPU and FPGA designs: two separate source bases, bit images, and boot mechanisms.

When integrating a 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. Also, 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 provides a high-level tool flow for CPU software and an FPGA design.

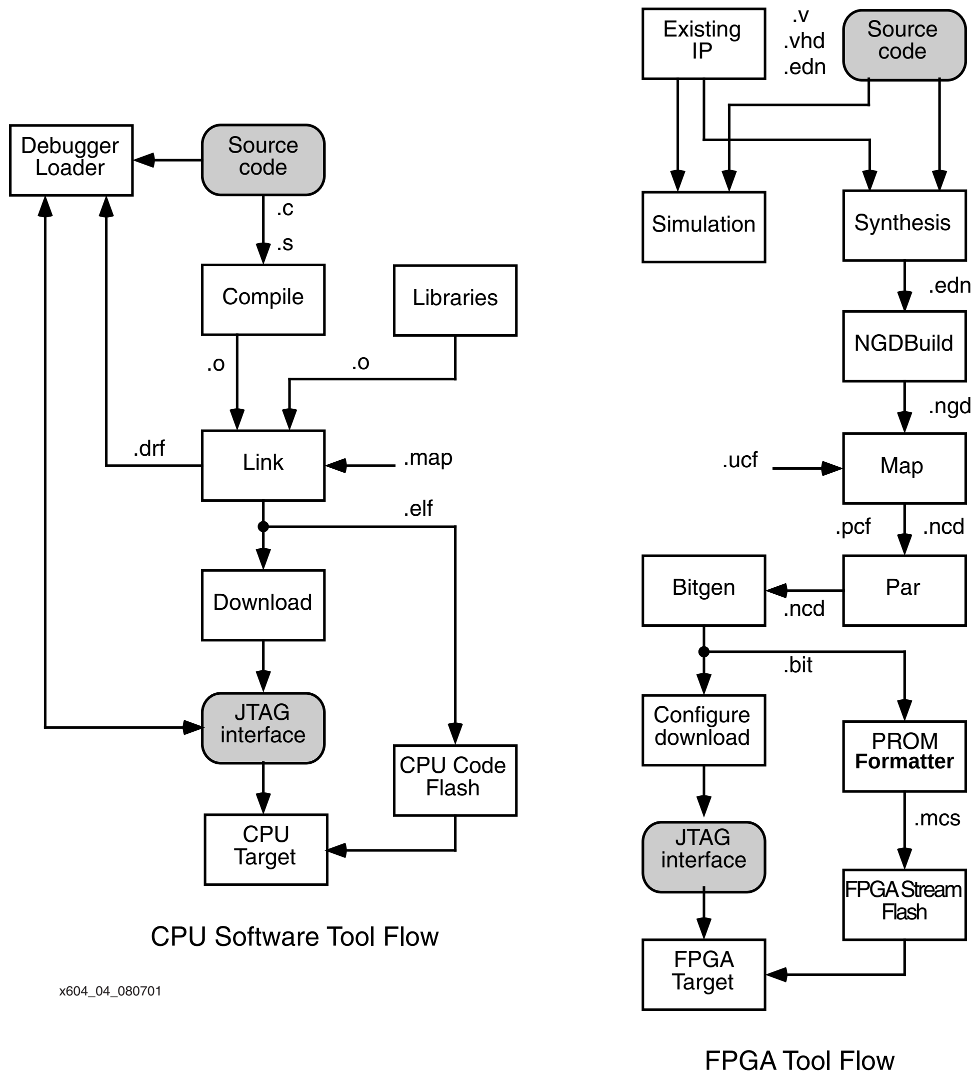

Figure 1-1: **HIgh Level Software and Hardware Tool Flows**

The following subsections describe the CPU software source code and FPGA design data flow.

CPU Software and FPGA Design Tool Flow

The following subsections describe how Data2MEM uses the CPU software source code and FPGA design code.

CPU Software Source Code

The flow on the left side of Figure 1-1, page 3 shows how the CPU software source code is used in the form of high-level C files and assembly-level S files. These files:

- Are compiled into O link files.
- The O files, with prebuilt O libraries, are linked together into a single executable code image.
- A MAP file is used in the link process to specify absolute address space locations, enabling the placement of executable code at specific address locations within system memory.

The output of the link process is either an ELF or a DRF file:

- The ELF contents can either be downloaded to a target directly through its JTAG debug port or programmed into the boot flash of the target.
- The executable portion of a DRF file can be downloaded to a target via a symbolic debugger, and the debug portion used to symbolically debug the executable code image.

FPGA Design

The flow on the right of Figure 1-1, page 3 shows how the FPGA source code is used in the form of V, VHD, and Electronic Data Interchange Format Netlist (EDN) files. These file go through the and the a design flow as follows:

- The files are used in various styles of hardware simulation or the files are synthesized into EDN intermediate files.
- A User Constraints File (UCF) and the intermediate EDN file are run through NGDBuild, MAP, and Place and Route (PAR) to produce a Native Circuit Description (NCD) file.
- Bitgen then 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.

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

Data2MEM Operational Overview

This section provides an overview of the data flow through Data2MEM by summarizing the design factors necessary when mapping CPU software code to Block RAM-implemented address spaces.

This overview 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 document.

The following are the design considerations for Block RAM-implemented address spaces:

- The Block RAMs come in fixed-size widths and depths, and 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 of data wide, for example, 32 or 64 bits (4 or 8 bytes) at a time.
- CPU bus accesses of multiple bytes of data 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 whole multiples of the CPU bus width. Bus bit (byte) lane interleaving is only allowed in the sizes supported by Virtex Block RAM port sizes. The data bus sizes by device family are:
    - 1, 2, 4, 8, and 16 bits for Spartan-II and the Virtex, Virtex-II, and Virtex-E devices.
    - 1, 2, 4, 8, 9, 16, 18, 32, and 36 bits for Spartan-3 and Virtex-II, Virtex-II Pro devices.
    - 1, 2, 4, 9, 18, and 36 bits for Virtex-4 devices.
    - 1, 2, 4, 9, 18, 36, and 72 bits Virtex-5 devices.
      Refer to "Block RAM Configurations by Device Family," page 8 for more detail.

  Note: When using parity, Data2MEM assumes the parity bits occupy the upper (Most Significant) bits of the device data bus. Refer to "Bit Lane Definitions (Memory Device Usage)" in Chapter 3 for more detail.
- 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 (byte) 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.

Given these considerations, refer to Figure 1-1, page 3.

- a 16 Kbyte address space from CPU address 0xFFFFC000 to 0xFFFFFFFF, constructed from the logical grouping of thirty-two 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*.

Note: Virtex, Virtex-E, Spartan2, and Spartan2E use 4-Kbit Block RAMs. Spartan-3, Spartan-3E, Spartan-3A, Virtex-II and Virtex-II Pro use 16-Kbit Block RAMs. Virtex-4 and Virtex-5 use 25-Kbit Block RAMs.

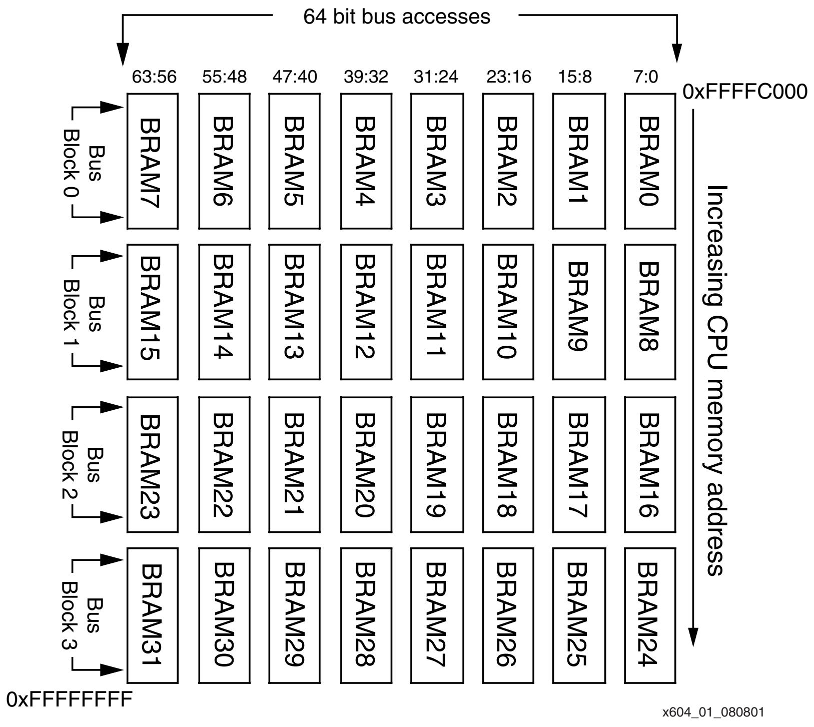

Figure 1-2: **Example Block RAM Address Space Layout**

The address space, or Address Block, shown inFigure 1-2, 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 then repeats for each successive bus block until the entire memory space is filled, or the input data is exhausted.

Note: At first, this filling order may seem counter-intuitive.; however, as described further in Chapter 3, "Block RAM Memory Map (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 (for example, in-house) 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), and any 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. For clarity in these examples, Figure 1-2, page 6 displays Block RAMs in ascending order. However, 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 Block RAM, however, can be configured in several widths, depending on hardware design requirements. The tables in "Block RAM Configurations by Device Family," page 8 specify the Virtex and Spartan Block RAM widths.
- Existing tools have limited configuration needs, so a simple command line interface will suffice. The Block RAM usage adds more complexity and requires a human-readable syntax to describe the mapping between address spaces and Block RAM utilization.

Block RAM Configurations by Device Family

Table 1-1: **Virtex, Virtex-E, Virtex-Em, Spartan-II, Spartan-IIE Block RAM Configurations**

Primitive	Data Depth	Data Width
RAMB4_S1	4096	1
RAMB4_S2	2048	2
RAMB4_S4	1024	4
RAMB4_S8	512	8
RAMB4_S16	256	16

Table 1-2: **Virtex-II and Virtex-II Pro, Spartan-3, Spartan-3A, Spartan-3E Block RAM Configurations**

Primitive	Data Depth	Data Cells	Parity Cells
		Data Width	Data Width
RAMB16_S1	16384	1	-
RAMB16_S2	8192	2	-
RAMB16_S4	4096	4	-
RAMB16_S9	2048	8	9
RAMB16_S18	1024	16	18
RAMB16_S36	512	32	36

Table 1-3: **Virtex-4 Block RAM Configurations**

Primitive	Data Depth	Data Cells	Parity Cells
		Data Width	Data Width
RAMB16	16384	1	-
RAMB16	8192	2	-
RAMB16	4096	4	-
RAMB16	2048	8	9
RAMB16	1024	16	18
RAMB16	512	32	36

Table 1-4: **Virtex-5 with 18 Kbit Block RAM Configurations**

Primitive	**Data Depth**	**Data Cells**	**Parity Cells**
		Data Width	**Data Width**
RAMB18	16384	1	-
RAMB18	8192	2	-
RAMB18	4096	4	-
RAMB18	2048	8	9
RAMB18	1024	16	18
RAMB18SDP	512	32	36

Table 1-5: **Virtex-5 with 36 Kbit Block RAM Configurations**

Primitive	**Data Depth**	**Data Cells**	**Parity Cells**
		Data Width	**Data Width**
RAMB36	32768	1	-
RAMB36	16384	2	-
RAMB36	8192	4	-
RAMB36	4096	8	9
RAMB36	2048	16	18
RAMB36	1024	32	36
RAMB36SDP	512	64	72

Chapter 2

Input and Output Files

This chapter describes the input and output files for Data2MEM. The following figure shows the range of files, and their input and output relationship to Data2MEM.

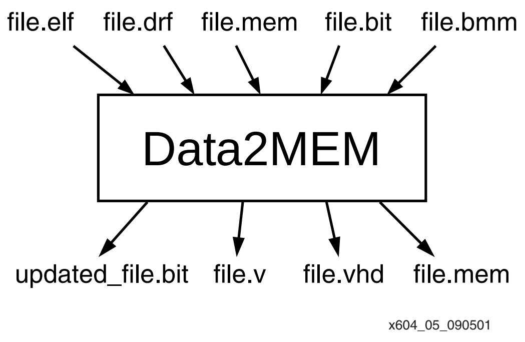

Figure 2-1: **Data2MEM Input and Output Files**

A description of each file type, and how it is consumed or produced by Data2MEM, is provided in the following sections:

- "Block RAM Memory Map (BMM) Files"
- "Executable and Linkable Format (ELF) Files"
- "Debugging Information Format DWARF (DRF) Files"
- "Memory (MEM) Files"
- "Bitstream (BIT) Files"
- "Verilog (V) Files"
- "VHDL (VHD) Files"
- "User Constraints Files (UCF)"

Block RAM Memory Map (BMM) Files

A Block RAM Memory Map (BMM) file is a simple text file that includes a syntactic description of how individual Block RAMs (BRAMs) constitute a contiguous logical data space. The Data2MEM uses the BMM file as input to direct the translation of data into the proper initialization form. A BMM file can be created by hand or using Data2MEM facilities that generate BMM file templates. The templates can then be customized to a specific design. A BMM file can also be created by means of automated scripting. Because it is a simple text file, a BMM file is editable. BMM files can contain free-form use of both `//` and `/*...*/` commenting styles. See Chapter 3, "Block RAM Memory Map (BMM) File Syntax" for format and syntax details.

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/linker tools. Refer to the appropriate software tools documentation for details on creating ELF files. Data2MEM uses ELF files as its basic data input form. Because ELF files are binary data, they are not directly editable. Data2MEM also provides some facilities for examining ELF file content. See Chapter 4, "Command Line Tool Usage" and Chapter 5, "Using the Integrated Implementation Tools" for additional information.

Debugging Information Format DWARF (DRF) Files

A Debugging Information Format (DRF, pronounced "dwarf") file is a binary data file that also contains the executable CPU code image along with debug information required by symbolic source-level debuggers. DRF files are produced by the same software compiler/linker tools as ELF files. Data2MEM inputs DRF files wherever ELF files can be used. Because DRF files are binary data, they are not directly editable. Data2MEM provides mechanisms to examine the content of DRF files. See Chapter 4, "Command Line Tool Usage" and Chapter 5, "Using the Integrated Implementation Tools" for additional information.

Memory (MEM) Files

A memory (MEM) file is a simple text file that describes contiguous blocks of data. MEM files are directly editable. 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 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. Therefore, hexadecimal values:

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

Would be interpreted as the values:

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

Note: The common **`0x`** hexadecimal prefix is not allowed. Using this prefix on MEM file hexadecimal values will be 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. The 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, in that an address specifier is only used once at the beginning of a contiguous block of data. The previous example would be 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 the following sections "Memory Files as Output" and "Memory Files as Input.")

A MEM file can have as many of these contiguous data blocks as required. There can be any size gap of address range between data blocks; however, no two data blocks can overlap an address range.

Memory Files as Output

Output MEM files are used primarily for Verilog simulations with third-party memory models. Therefore, the format follows industry standard usage on the following key points:

- 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 it is undefined. See Chapter 3, "Block RAM Memory Map (BMM) File Syntax" for use of the `OUTPUT` keyword to generate output MEM files.

Memory Files as Input

Input MEM files have format restrictions that do not conform to the industry standard. Please note the following key differences:

- 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.

  Note: Odd-length data values are rounded up to an even eight-bit size, with the upper four bits assumed to be zero.
- If an address gap exists between two contiguous blocks of data, the address gap is assumed to be a non-existent 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 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. Additionally, hexadecimal format only allows values to be defined in even 4 bit nibble values. Hence, 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 will discard any additional bits added by the hexidecimal 4 bit width restriction.

For example, an 18 bit data value, 0x23A24, can only be specified in hexadecimal 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 Block RAM's data bus is 18 bits wide, it should discard the two data bits. Similarly, a 9 bit data value, 0x1D4, would have the top three data bits discarded. This "discarding" process will also be used with non-parity Block RAM data widths that are 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 a FPGA device. Data2MEM can directly replace the Block RAM data in BIT files without the intervention of 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. Refer to Xilinx implementation tools documentation for the details on creating BIT files. Because BIT files are binary data, they are not directly editable. Data2MEM also provides some facilities for examining the BIT file content. See Chapter 4, “Command Line Tool Usage” for details.

Verilog (V) Files

A Verilog (V) file is a simple text file output by Data2MEM. It contains `defparm` records to initialize Block RAMs. This file is used primarily for pre- and post-synthesis simulation. Because a V file is a simple text file, it is directly editable. However, because a V file is a generated file, editing is not advised. Data2MEM allows the free-form use of both `//` and `/*...*/` commenting styles. See Chapter 4, “Command Line Tool Usage” for details.

VHDL (VHD) Files

A VHDL (VHD) file is a simple text file output by Data2MEM. It contains `bit_vector` constants to initialize Block RAMs. These constants can then be used in “generic maps” to instance an initialized Block RAM. VHD files are used primarily for pre- and post-synthesis simulation. Because a VHD file is a simple text file, it is directly editable. However, because this file is a generated file, editing is not advised. Data2MEM allows the free-form use of both `//` and `/*...*/` commenting styles. See sections “Command Line Tool Usage” for usage details.

User Constraints Files (UCF)

A User Constraints File (UCF) is a simple text file output by Data2MEM. It contains `INST` records to initialize Block RAMs. Because a UCF is a simple text file, it is directly editable. However, because a UCF is a generated file, editing is not advised. Data2MEM allows the free-form use of both `//` and `/*...*/` commenting styles. This file type is supported for legacy workflows. Its use for new designs or workflows is discouraged.

Chapter 3

Block RAM Memory Map (BMM) File Syntax

This chapter provides details of the syntax used in Block RAM Memory Map (BMM) files in the following sections:

- "Features"
- "ADDRESS_MAP Definitions (Multiple Processor Support)"
- "ADDRESS_SPACE Definitions (a Logical Address Space)"
- "BUS_BLOCK Definitions (Bus Accesses)"
- "Bit Lane Definitions (Memory Device Usage)"
- "Combined Address Spaces"

Appendix A, "Example BMM file," shows the text-based syntax created to describe the organization of Block RAM (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 Figure 1-2, page 6.

Features

BMM is oriented toward human readability and is similar in the following ways to high-level computer programming languages:

- 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. As with any high-level computer language, allowing plain-text documentation embedded within the file contents preserves knowledge and promotes coherence. BMM files allow the free-form use of comment blocks anywhere within the content of the file.
- Implied algorithms. While it is easier to think of data transpositions in data-associative terms, computers express data transpositions in purely algorithmic terms. BMM allows you to specify the data transposition in semi-graphical terms, while alleviating the need to specify the exact details of the address-to-Block RAM algorithm. The computer then infers the algorithm details for the desired mapping.

It is important to be aware of the following notational details:

- Keywords are case-sensitive and are always uppercase.

- Appendix A, “Example BMM file” shows a recommended indenting style. However, this style is for clarity only. White space is ignored except where it delineates items or keywords.
- Line endings are ignored. As many items as desired can appear 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.
  - `//` means 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.

See Appendix B, “BMM Modified Backus-Naur Form Syntax.” to understand the modifications to Backus_Naur form syntax used in BMM.

ADDRESS_MAP Definitions (Multiple Processor Support)

Data2MEM now includes better support of multiple processors. This functionality is enabled with the addition of two new keywords: `ADDRESS_MAP` and `END_ADDRESS_MAP`, which are used in the form:

```
ADDRESS_MAP map_name processor_type processor_ID

     ADDRESS_SPACE space_name mtype [start:end]
          .
          .
     END_ADDRESS_MAP;
     .
     .
END_ADDRESS_MAP;
```

These new keywords surround all the `ADDRESS_SPACE` definitions that belong to the memory map for a single processor. The `map_name` is an identifier that refers to all the `ADDRESS_SPACE`s in an `ADDRESS_MAP`. The `processor_type`, as its name suggests, specifies processor type. Currently, only the PowerPC® 405 processor type is allowed. The `processor_ID` is used by iMPACT as a JTAG ID to download external memory contents to the proper processor.

Each processor has its own `ADDRESS_MAP` definition. An `ADDRESS_SPACE` name must only be unique 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 now take on two new forms and can be substituted wherever an `ADDRESS_SPACE` name was used previously. First, 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`. Second, the address tag name can be shortened to just the `ADDRESS_MAP` name, which confines data translation to only those `ADDRESS_SPACE`s within the named `ADDRESS_MAP`. This is most conveniently used for sending 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. However, those `ADDRESS_SPACE`s are assumed to belong to an unnamed `ADDRESS_MAP` definition of type `PPC405` and processor `ID 0`. Address tags for these `ADDRESS_SPACE`s 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`s that have matching address ranges.

ADDRESS_SPACE Definitions (a Logical Address Space)

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

```
ADDRESS_SPACE ram_cntlr RAMB4 [start_addr:end_addr]
     .
     .
END_ADDRESS_SPACE;
```

The `ADDRESS_SPACE` and `END_ADDRESS_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 Figure 1-2, page 6. A BMM file can contain multiple `ADDRESS_SPACE` definitions, even for the same address space, as long as the name for each `ADDRESS_SPACE` is unique.

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

- `RAMB4`
- `RAMB16`
- `RAMB18`
- `RAMB32`
- `RAMB36`
- `MEMORY`
- `COMBINED`

The `RAMB4` keyword defines the memory device as a 4-Kbit Block RAM found in Virtex™ and Virtex-E parts.

In Spartan™-3, Virtex-II, and Virtex-II Pro devices, the `RAMB16` keyword defines the memory as a 16-Kbit Block RAM without parity included, and `RAMB18` keyword defines the memory space as an 18-Kbit Block RAM using parity

Additionally, the Virtex-5 can use the `RAMB32` keyword that defines the Block RAM memory size and style as a non-parity memory and `RAMB36` keyword that defines a36-Kbit Block RAM using parity memory.

The correct keyword must be used 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`.

Refer to the "Combined Address Spaces," page 22 for detail on the `COMBINED` keyword.

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. The composition of the blocks are as follows:

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

Each Bus Block brackets those Block RAM Bit Lane definitions that are accessed by a parallel CPU bus access. In the case of Appendix A, "Example BMM file" there are four, which correspond to the four Bus Block rows in Figure 1-2, page 6.

The order in which the Bus Blocks are specified defines what 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 Appendix A, "Example BMM file," the first Bus Block would occupy CPU addresses 0xFFFFC000 to 0xFFFFCFFF. This is the same as the first row of Block RAMs in Figure 1-2, page 6. The second Bus Block would occupy CPU addresses 0xFFFFD000 to 0xFFFFDFFF, which represents the second row of Block RAMs shown in the figure. This pattern repeats in ascending order until the last Bus Block.

Note: The top-to-bottom order in which Bus Blocks are defined also controls the order in which Data2MEM will fill them with data.

Bit Lane Definitions (Memory Device Usage)

A Bit Lane definition selects which bits in a CPU bus access are assigned to which 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 the hierarchy path of the Block RAM, as used in the Hardware Description Language (HDL) design. The syntax is as follows:

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

Additionally, 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 should be connected to the normal data bits of the Block RAM, and bits 17 and 16 should be connected to the parity bits of the Block RAM.

Note: Normally the bit numbers are given as shown in the order above, `[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 will bit-reverse the Bit Lane value 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 what 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 case of Figure 1-2, page 6, the most significant Bit Lane is BRAM7, and the least significant Bit Lane is BRAM0. As seen in Appendix A, "Example BMM file," this corresponds with the order in which the Bit Lanes are defined.

It is also important to understand how Data2MEM inputs data. Data is taken from data input files in Bit Lane sized chunks, from the most significant value first to the least significant. If the first 64 bits of input data were 0xB47DDE02826A8419 then the value 0xB4 would be the first value to be set into a Block RAM.

Given the Bit Lane order, BRAM7 would be set to 0xB4, BRAM6 to 0x7D, and so on. This would repeat until BRAM0 was 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. Figure 3-1, page 21 expands the first Bus Block of Figure 1-2, page 6 to illustrate this process.

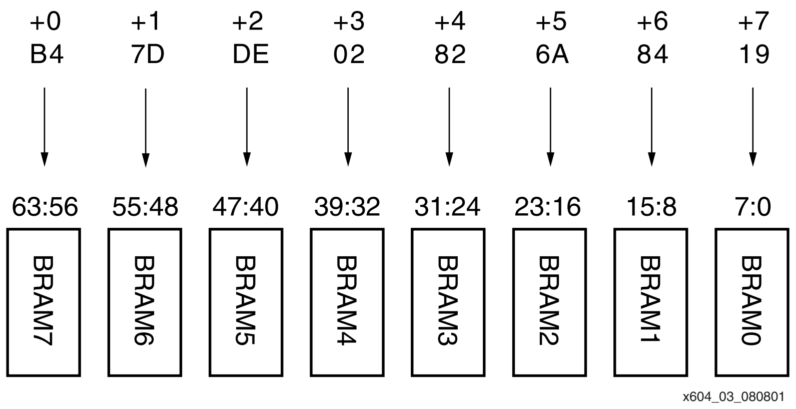

Figure 3-1: **Bit Lane Fill Order**

Note: 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 `RAMB4` or `RAMB16` block RAM devices, the physical row and column location within the FPGA can be indicated. The following are examples of the physical row and column location:

```
    top/ram_cntlr/ram0 [7:0] LOC = R3C5;
```

or

```
    top/ram_cntlr/ram0 [7:0] PLACED = R3C5;
```

Use the `LOC` keyword to hard-locate the corresponding block RAM. In this case, at row 3 and column 5, within the HDL design. The `PLACED` keyword is used by the Xilinx® implementation tools when creating a *back-annotated* BMM file. The example above indicates that the implementation tools located the corresponding Block RAM at row 3 and column 5, during BIT file generation. (See Chapter 5, "Using the Integrated Implementation Tools" for more information on back-annotated BMM files.) These definitions are inserted after the bus-bit values and the terminating semicolon. The location constraints also override any existing constraints in a UCF.

Note: The RxCx syntax is used with the `RAMB4` keyword for Virtex and Virtex-E parts. All other devices use the XxYx syntax.

An `OUTPUT` keyword can be used 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 you can see in Appendix A, "Example BMM file," MEM files are created for all the block RAMs in the second Bus Block.

Beyond the syntax of Bit Lane and Bus Block definitions, a number of constraints must also be observed:

- 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, and 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 a violation is detected.

Combined Address Spaces

The BMM address space is synonymous to 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 code 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_ADDRESS_SPACE;
```

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

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 user 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.

Previously, the 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 new keyword, COMBINED.

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_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;
```

The distinctions between the two code examples are:

- The use of the new 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_ADDRESS_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.

Lastly, 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 automatically continues with the next address range.

An additional benefit of this option is, 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. This means that a *logical* address space can be a mix of *physical* memory types, which provides a greater range of flexibility in memory options.

Command Line Tool Usage

This chapter describes the various categories of command line functionality and shows typical uses. The following sections are included in this chapter:

- "Block RAM Memory Map (BMM) File Syntax Checking"
- "Data File Translation or Conversion"
- "Data File Translation with Tag Name Filtering"
- "Bitstream (BIT) File Block RAM Replacement"
- "Examining BIT and ELF File Contents"
- "Miscellaneous Functionality"

Block RAM Memory Map (BMM) File Syntax Checking

The `-bm` option allows a designer to syntax-check a BMM file. It is invoked as:

```
data2mem -bm my.bmm
```

Based on this example, Data2MEM would parse the BMM file `my.bmm` and report any errors or warnings. If no output is given, the BMM file is correct. Only the BMM syntax is checked for correctness. It is still up to you to ensure that the BMM file matches the logic design.

Data File Translation or Conversion

In combination with the `-bm` option, the `-bd` and `-o` options are used to transform Executable and Linkable Format (ELF) or memory (MEM) data files into a different format. Principally, data files are converted into Block RAM initialization files for Verilog and VHDL, or User Constraints File (UCF) block RAM initialization records. A conversion to all three formats would be invoked as:

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

This would yield the files `output.v`, `output.vhd`, and `output.ucf`. While only one data file is shown here, as many `-bd datafile` pairs as needed can be given. These files can then be incorporated directly into the design source file set, or they can be 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. This would be invoked as:

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

The file `code.mem` would contain a text-based version of the contents of the binary ELF file. This is useful for making patches to ELF files for which the source 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. With a command line invoked as:

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

and a BitLane that appears as:

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

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

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

Data File Translation with Tag Name Filtering

Data file translation can be further controlled with tag or Address Block name filtering. By listing a set of Address Block names with each `-bd` option, data translation is confined only to that set of Address Blocks. A `-bm` option might be modified as:

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

In this way, data translation only takes place for the Address Blocks `mem1` and `mem2`, even if data in `code.elf` matches another Address Block. This allows you to *steer* different data contents to Address Blocks that may have the same address range. Alternatively, this facility allows data translation to be restricted to a portion of the design, leaving the rest of the design untouched.

Note: Using tag name filtering implicitly invokes the `-i` option to turn off address space mismatch errors.

Bitstream (BIT) File Block RAM Replacement

Data2MEM provides the facility to iterate new block RAM data into a BIT file without the need to rerun the Xilinx® implementation tools. In conjunction with a new ELF and BMM file, Data2MEM updates the block RAM initialization in a BIT file image and outputs a new BIT file. In addition, tag filtering can be included. This facility is invoked as:

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

Working from this example, the result would be a new BIT file called `new.bit`, with the appropriate block RAM contents replaced with the contents of `code.elf`.

Note: For proper operation, 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. See Chapter 5, "Using the Integrated Implementation Tools" for more information.

The process yields a new BIT file that improves speed significantly (from 100 to 1000 times) over rerunning the implementation tools. This facility is meant primarily as a means to include new CPU software code into a design when the logic portion of the design is not changing. Most often this will be used by a software developer who has no access (nor understanding) of the Xilinx implementation tools.

Examining BIT and ELF File Contents

Data2MEM provides the ability to examine, or *dump*, the contents of ELF and BIT data files. The dump content is in a text hexadecimal format that is pertinent to the input data file and is printed to the console. Also, the `-d` option has two optional parameters, `e` and `r`, that change what input data file dependent information is displayed.

ELF dumps are invoked as:

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

The dump shows the contents of each section within the ELF file. Using the `e` option will display additional information about each section. Using the `r` option will include some redundant ELF header information.

Note: ELF files contain much more data than that used for Data2MEM data translation (symbols, debug, and so on.). Only those sections labeled "Program header record" are considered by Data2MEM for data translation.

BIT dumps are invoked as:

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

Each bit stream command is decoded and displayed. Those commands that contain bit field flags have each bit field described. Commands that contain non-block RAM data chunks are displayed 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 are also useful for comparing binary ELF and BIT files with simple, human-friendly text tools.

Miscellaneous Functionality

Data2MEM has two other options to control its behavior.

The `-i` option tells Data2MEM to ignore any data in an ELF or MEM file that is outside any Address Block within the BMM file. This allows data files to be used that have much more data in them than the BMM file recognizes. For example, Data2MEM can use a master design code file as an ELF data file, though only a small portion of that file is data destined to be ELF code for block RAM memories.

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 it contains initializations of all zeroes. If this option is not used, only Address Spaces that receive transformed data will be output.

Chapter 5

Using the Integrated Implementation Tools

This chapter describes how Data2MEM functionality integrates with the Xilinx® implementation tool flow. 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 sub-set of Data2MEM options in NGDBuild, Bitgen, Netgen, and FPGA Editor. Figure 5-1 illustrates the software flow and the file dependencies.

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.

The following sections in this chapter describe each option and its effect in the software flow:

- "Using NGDBuild"
- "Using MAP and PAR"
- "Using Bitgen"
- "Using Netgen"
- "Using FPGA Editor"
- "Restrictions"

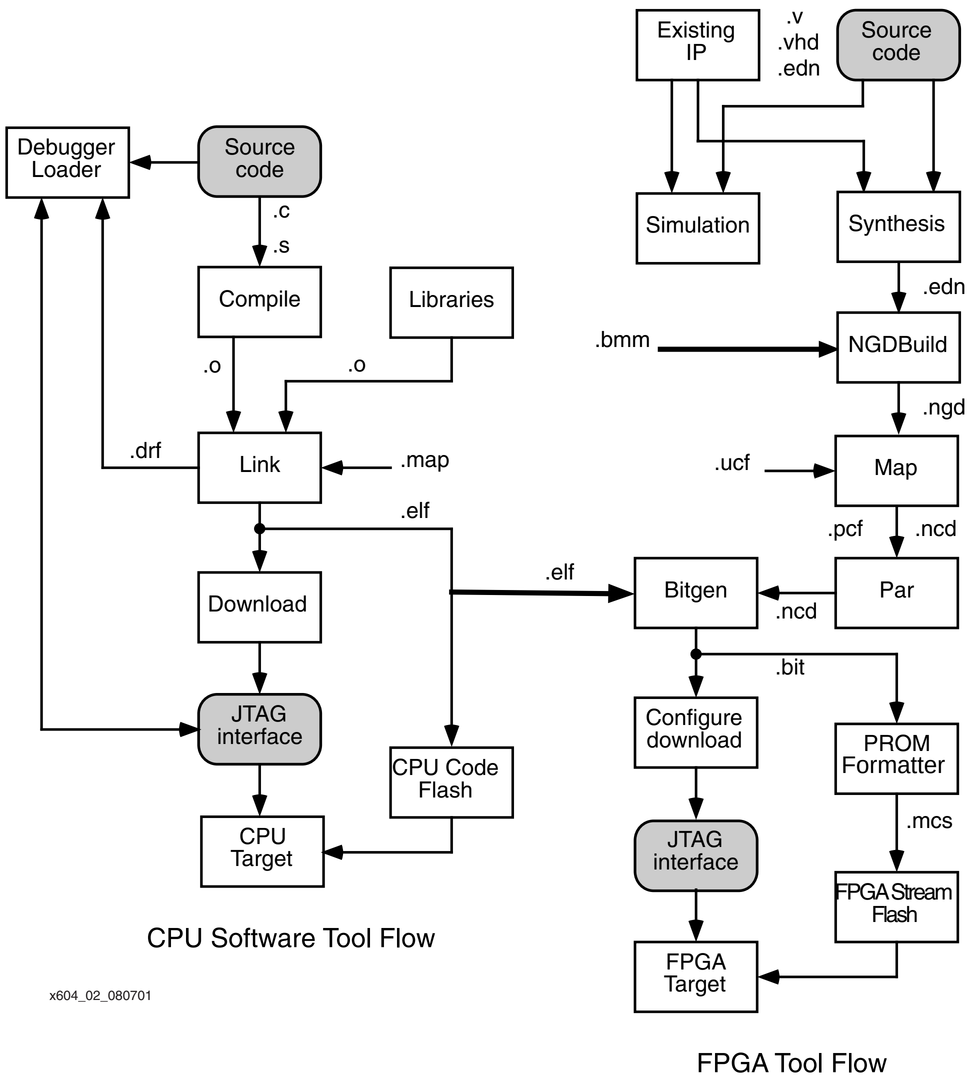

Figure 5-1: **Integrated Data2MEM and Implementation Tool Flow**

Using NGDBuild

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

Usage

The `-bm` option allows you to specify the name and path of the BMM file. If the BMM file name is the same as the design name and is located in the same directory as the design netlist, the BMM file is loaded by default. If the BMM file is added to an ISE™ project, ISE tracks changes to the BMM file and re-implements the design when necessary.

Functionality

NGDBuild creates the `BMM_FILE` property in the Native Generic Database (NGD) file to let following tools know a BMM file design is being used. The BMM file is syntax-checked, and NGDBuild validates that the block RAMs referenced in the BMM file actually exist in the design. (Syntax checking of the BMM file can also be done by running the command line version of Data2MEM.) Also, any block RAM placement constraints that appear in the BMM file are applied to the corresponding block RAM.

Note: The `-bm` switch is supported in ISE. Refer to the for how to set this option so that it is invoked in NGDBuild.

Using MAP and PAR

There are no command line or functionality changes to MAP or Place and Route (PAR). However, incorrectly connected block RAM components are trimmed by MAP. You are responsible for correctly connecting the block RAM components. Check the MAP report "Section 5 - Removed Logic" to see if any block RAM components were removed from the design.

Using Bitgen

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

Usage

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

Functionality

Bitgen passes the `-bd` switch with the `<filename>` and any tag information to Data2MEM. Data2MEM processes the BMM file specified during the NGDBuild phase, and 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 Circuit Description (NCD) 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 to create a `<BMMfilename>_bd.bmm` file, a back-annotated BMM file.

Note: If Bitgen is invoked with a NCD file that contains a `BMM_FILE` 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.

Note: The `-bd` switch is supported in ISE. Refer to the http://www.xilinx.com/products/design_resources/design_tool/index.htm for how to set this option so that it is invoked by Bitgen.

Using Netgen

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

Usage

The **-bd** switch specifies the path and name of the ELF file used to populate the block RAMs specified in the BMM file. The address information contained in the ELF file allows Data2MEM to determine the `ADDRESS_SPACE` in which to place the data.

Functionality

`Netgen` passes the `-bd` switch with the `<elf_filename>` to Data2MEM. Data2MEM processes the BMM file specified during NGDBuild and the ELF file is used to create the block RAM initialization strings for each of the constrained block RAMs. The initialization strings are then used to update the NCD file before the BIT file is created.

Placement information of the block RAM is provided from the NCD file to Data2MEM. This information is used to create the `<bramfilename>_bd.bmm` file. This file contains the placement information for all BRAM constrained or unconstrained. This is necessary to enable the use of the command line version of Data2MEM.

```
Option: -bx
Syntax: -bx [filepath]
```

Usage

The **-bx** switch specifies the file path to output individual memory device MEM files for performing HDL simulations. From the contents supplied in the `-bd` switches. The `-bd` switch must be supplied when using `-bx` option.

Functionality

`Netgen` passes the `-bx` switch with the optional *`filepath`* to Data2MEM. Date2MEM 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 will not be generated automatically.

The resulting netlist output will have each Block RAM instance annotated with a `INIT_FILE` parameter to indicate the MEM file with which to initialize the Block RAM. During subsequent simulations, the memory file in the `INIT_FILE` parameter will be used to initialize the contents of the Block RAM.

Note: This functionality is currently only avaible to Virtex-4 and Virtex-5 devices.

Updated MEM files can then be created by invoking Data2MEM alone (refer to the -bx command option in). This avoids the necessity of rerunning Netgen to generate updated MEM files.

Note: The -bd and `-bx` switches are supported in ISE. Refer to thttp://www.xilinx.com/products/design_resources/design_tool/index.htm f or how to set this option for use in Netgen.

Using FPGA Editor

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

Usage

The `-bd` switch specifies the path and file name of the ELF file used to populate the block RAMs specified in the BMM file. The address information contained in the ELF file allows Data2MEM to determine the `ADDRESS_SPACE` in which to place the data.

Functionality

The Block RAMs specified in the BMM file are marked as read-only in FPGA Editor. 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.

Note: The `-bd` switch is supported in ISE. Refer to the ISE documentation on the Xilinx website, http://www.xilinx.com/products/design_resources/design_tool/index.htm for information on how FPGA Editor can invoke this option.

Restrictions

The restrictions on integrated implementation tool usage of Data2MEM functionality are described as follows:

- 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, or Netgen.
- Block RAMs specified in the BMM file may 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 HDL hardware design.

Chapter 6

Integrated iMPACT Tool Usage

This chapter describes the iMPACT tool integration with Data2MEM and describes the process flow.

Overview

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

1. You call up the configuration dialog in iMPACT. There, the Block Ram Memory Map (BMM) and Executable and Linkable Format (ELF) files are chosen.
2. You then choose tag names to associate with the ELF files, which confine ELF file translation to just the chosen tag name `ADDRESS_SPACE`s.
3. A boot address can be entered for each destination processor.

iMPACT Tool Process Flow

- The iMPACT tool reads in the BIT file and passes 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`s in the BMM and replaces any block RAM contents in the BIT file memory image. The memory image is handed back to iMPACT.
- The iMPACT tool configures the updated BIT file memory into the FPGA device and halts any processors.
- The iMPACT tool then requests any external memory data from Dat2MEM. 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). The iMPACT tool 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.
- The iMPACT tool 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.

- The iMPACT tool 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 Field Programmable Gate Array (FPGA) device. The iMPACT tool can then translate the SVF file into an ACE file, to use with SystemACE. This puts the configure stream into a shippable form, with complete configure, block RAM, and external memory initialization.

For the process to work properly, 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.

Command Line Option Reference

Overview

The Data2MEM command line syntax is as follows:

```
data2mem
<-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 ]>
```

The following section provides a list of the options along with a description.

Command Line Options and Descriptions

Table 7-1: **Command Line Options**

Command	Description
-bm *filename*	Name of the input Block Memory Map (BMM) file. If the file extension is missing, a BMM file extension is assumed. If this is option not unspecified, 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.
-bd *filename*	Name of the input ELF or MEM files. If the file extension is missing, .elf 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_ADDRESS_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`s 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_ADDRESS_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 seperates 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 preceed 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.

Table 7-1: **Command Line Options *(Continued)***

Command	Description
-bx *filepath*	File path to output individual memory device MEM files for performing HDL simulations. If an `OUTPUT` keyword exists on a BitLane, 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 `TagNames` 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` 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_ADDRESS_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`s 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_ADDRESS_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 seperates 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 preceed 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.

Table 7-1: **Command Line Options *(Continued)***

Command	Description

-o u|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.
	m = MEM file format, a .mem file extension (obsolete).
	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.

-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 initializations of 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** m 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 availible 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 = Alpha-numeric name of a User Defined Memory Device. • MSIZE = Hex byte size of the User Memory Device. I.e., 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.

Table 7-1: **Command Line Options *(Continued)***

Command	Description
`-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 =Alpha-numeric name of the Processor Map. PTYPE = Alpha-numeric name of the processor type in the Address Map. Legal processor types consists of 'PPC405', 'MICROBLAZE',and 'PICOBLAZE'. 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 = Alpha-numeric 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. ASTART = Hex address the Address Space starts from. I.e., 0xFFF00000 BWIDTH = Numeric bit width of the bus access for the Address Space. 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. MTYPE = The memory type the Address Range is construct of. Legal memory types are 'RAMB4', 'RAMB16', 'RAMB18', and any User Defined Memory Device. BSIZE = Hex byte size of the Address Range. I.e., 0x1FFFF. DWIDTH = Numeric bit width of each bitLane 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.

Table 7-1: **Command Line Options *(Continued)***

Command	Description
`-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. ASIZE = Hexidecimal byte size of the Address Template. For example, 0x1FFFF. BWIDTH = Numeric bit width of the bus access for the Address Template. 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. MTYPE = The memory type the Address Range is construct of. Legal memory types are:RAMB4', 'RAMB16', 'RAMB18', and any User Defined Memory Device. BSIZE = Hex byte size of the Address Range. I.e., 0x1FFFF. DWIDTH = Numeric bit width of each bitLane 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 of the root instance name. Values start at zero. In addition, a base instance path can begin with a '*/' syntax. When the template is expanded to an instance, instance IROOT and IBASE are joined together to form a complete instance path.
-mf i 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 theinstance items can be applied to an address template, to createa new address instance. TNAME = Alpha-numeric name of the Address Template. INAME = Alpha-numeric name of the Address Instance. ASTART = Hex address the Address Space starts from. I.e., 0xFFF00000 IROOT = Root alpha-numeric hierarchy/part instance name applied to each bitLane 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™ part. If this is unspecified, an `xcv50` part is assumed. Use the `-h` option to obtain the full supported part name list.

Table 7-1: **Command Line Options *(Continued)***

Command	Description
-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.
-q 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	Print help text, plus supported part name list.

XILINX®

Appendix A

Example BMM file

```
/*****************************************************
*
* FILE : example.bmm
*
* Define a BRAM map for the RAM controller memory space. The
* address space 0xFFFFC000 - 0xFFFFFFFF, 16k deep by 64 bits wide.
*
**************************************************** /

ADDRESS_SPACE ram_cntlr RAMB4 [0xFFFFC000:0xFFFFFFFF]

    // Bus access map for the lower 4k, CPU address 0xFFFFC000 - 0xFFFFCFFF
    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 4k, CPU address 0xFFFFD000 - 0xFFFFDFFF
    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 4k, CPU address 0xFFFFE000 - 0xFFFFEFFF
    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 4k, CPU address 0xFFFFF000 - 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_ADDRESS_SPACE;
```

BMM Modified Backus-Naur Form Syntax

```
Address_block_keyword::="ADDRESS_SPACE";

End_address_block_keyword::="END_ADDRESS_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::="RAMB4" |
                    "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_address_block_keyword ";" ;
```