Data2BRAM

Data2BRAM is compatible with the following families:

- Virtex-II Pro™
- Virtex™/-II/-E
- Spartan-II™/-E

This document describes how the Data2BRAM software tool automates and simplifies setting the contents of BRAM cells on Virtex™ devices. It also shows how this is used with the 32-bit CPU on the single-chip Virtex-II Pro devices.

The chapter contains the following sections:

- "Introduction"
- "Feature Summary"
- "Input and Output Files"
- "Usage Overview"
- "Process Overview"
- "BMM (Block RAM Memory Mapping) File Syntax"
- "Command Line Tool Usage"
- "Integrated Xilinx Implementation Tool Usage"
- "Command Line Option Reference"

Introduction

Data2BRAM is fundamentally a data translation tool. It translates contiguous blocks of data across multiple Block RAMs, that constitute a contiguous logical address space . With the combination of Virtex series devices, and an embedded CPU into a single chip, Data2BRAM incorporates CPU software images into FPGA bitstreams. This allows CPU software to be executed from Block RAM-built memory, from within a FPGA bitstream. This presents a powerful and flexible means of merging parts of CPU software, and FPGA design tool flows. Lastly, Data2BRAM can also be used as a simplified means for initializing Block RAMs for non-CPU designs.

While Data2BRAM has automated a complicated process into significantly simplified technique, it also accomplishes the following goals:

- Affect existing tool flows as little as possible, for both FPGA and CPU software designers.
- Limit the time delay one tool flow imposes on the other for testing changes or fixing verification problems.
- Isolate the process to a single step or as few steps as possible.
- Reduce or eliminate the requirement for one tool flow user (for example, CPU software or FPGA designer) to learn the other tool flow steps and details.

Data2BRAM is supported on the following platforms:

- Solaris 2.8
- Windows 2000, with SP2 or higher
- Windows XP, and Windows XP Home

Feature Summary

Data2BRAM provides 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 (byte) lane interleaving.

- Easily adapts to the multiple data widths available from BRAM models.
- Reads Executable and Linkable Format (.elf) files, or DWARF Debugging Information Format (.drf) files (DWARF is a play on the ELF file format name) 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) files as arbitrary input for Block RAM contents. This simple text format can be either hand or machine generated.
- Can produce formated text dumps of the contents of Bit (.bit), Executable and Linkable Format (.elf), and DWARF Debugging Information Format (.drf) files.
- Produces .v (for Verilog) and .vhd (for VHDL) initialization files for pre- and post-synthesis simulation.
- Seamlessly integrates initialization data into post-place and route simulations.
- Produces Memory definition (.mem) files for Verilog simulations with third-party memory models.
- Can replace the contents of Block Ram memory in Bit (.bit) files directly without intervention of any other Xilinx implementation tool. This avoids lengthly implementation tool runs.
- Can be invoked as a command line tool, or as an integrated part of Xilinx implementation tool flow.
- Recognizes all common types of text line endings (Windows, Unix, etc.), and uses them interchangeably.
- Allows the free-form use of "//" and "/*...*/" commenting syntax in all text input files.

Input and Output Files

Data2BRAM utilizes a number of input and output files. Figure 1 portrays the range of files, and their input/output relationship to Data2BRAM. Below is a description of each file type, and how they are consumed or produced by Data2BRAM.

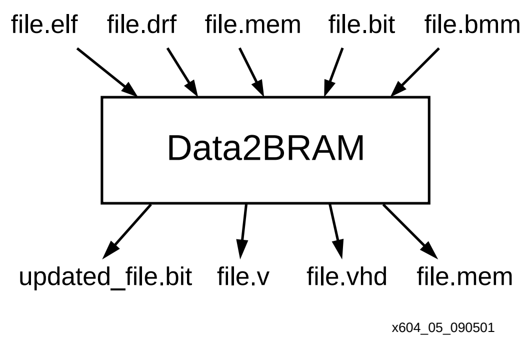

Figure 1 Data2BRAM Input and Output Files

Block RAM Memory Map (.bmm) files

A BMM file (Block RAM Memory Map) is a simple text file that has syntactic description of how individual Block RAMs constitute a contiguous logical data space. This is a fundamental input file which Data2BRAM uses to direct the translation of data into the proper initialization form. A BMM file can becreated by hand, or with Data2BRAM facilities to generate BMM file templates. These templates can then be customize to a specific design. A BMM file can also be created by automated scripting means. Since a BMM file is a simple text file, it is directly editable. BMM files can contain free-form use of both "//" and "/*...*/" commenting styles. See section "BMM (Block RAM Memory Mapping) File Syntax" for format and syntax details.

Executable and Linkable Format (.elf) files

An .elf file (pronounced "elf") is a binary data file that contains an executable CPU code image, ready for running on a CPU. These files are produced by software compiler/linker tools. Please refer to the proper software tools documentation for the details on creating .elf files. Data2BRAM uses .elf files as it's basic data input form. Since .elf files are binary data, they are not directly editable. Data2BRAM also provides some facilities for examining the content of .elf files. See sections "Command Line Tool Usage", and "Integrated Xilinx Implementation Tool Usage" for usage details.

Debugging Information Format DWARF (.drf) files

A .drf file (pronounced "dwarf") is a binary data file that also contains the executable CPU code image, plus debug information required by symbolic source-level debuggers. These files are produced by the same software compiler/linker tools as .elf files. Data2BRAM will input .drf files wherever .elf files can be used. Since .drf files are binary data, they are not directly editable. Data2BRAM provides some facilities for examing the content of .drf files. See sections "Command Line Tool Usage", and "Integrated Xilinx Implementation Tool Usage" for usage details.

Memory (.mem) files

A .mem file (memory) is a simple text file that describes contiguous blocks of data. Since a .mem file is a simple text file, it is directly editable. Data2BRAM allows the free-form use of both "//" and "/*...*/" commenting styles. Data2BRAM uses .mem files for both data input and output.

The format of .mem files is an industry standard, and consists of two basic elements; hex address specifier and hex data values. An address specifier is indicated by a "@" character followed the hex address value. There are no spaces between the "@" character and the first hex character.

Hex data values follow the hex address value, separated by spaces, tabs, or carriage-return characters. Data values can consist of as many hex characters as desired. However, when a value has an odd number of hex characters, the first hex character is assumed to be a "0". Therefore, hex values:

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

Would be interpreted as the values:

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

Note The common "0x" hex prefix is not allowed. Using this prefix on .mem file hex values will be flaged 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...
```

Data2BRAM requires a less redundant format in that an address specifier is only specified 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 from its distance from the previous address specifier. However, the derived addresses depends whether the file is being used as an input or output. See the description of the differances between input and output memory files below.

A .mem file may 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 three key points.

1. All data values must be the same number of bits wide, and must be the same width as expected by the memory model.
2. Data values reside within a larger *array* of values, starting at zero. An address specifier isn't a true *address*, but rather, 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 hex location (given that indexes start at zero), within an array of 16 bit data values:

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

3. If an address gap exists between two contiguous blocks of data, then the data between the gaps still logically exists, but it is just undefined. See section "BMM (Block RAM Memory Mapping) File Syntax" for usage 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. There are six key differences to understand.

1. White space between adjacent data values is ignored. Rather, all of the values in a contiguous blocks of data are treated as a continuous stream of bits. Data2BRAM breaks up the bit stream into data values according to the width the target Block RAMs are configured. White space between adjacent data values is used solely for readability.
2. 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. Rather, the specifier is any number that matches a BMM address space.
3. Derived addresses for successive data values depends on a value's byte length, despite the fact that address specifiers are not specifically CPU memory address. A eight bit value increments the next derived address by one, a sixteen value by two, thirty two bit value by four, and so forth.

   Note As was stated above, odd length data values are rounded up to an even eight bit size, with the upper four bits assumed to be zero.
4. If an address gap exists between two contiguous blocks of data, then the address gap is assumed to be a non-existent memory.
5. No two contiguous blocks of data can overlap an address range.
6. A contiguous block of data must fit within a single address space range defined in a BMM file.

Bit (.bit) files

A .bit file (Bit Stream) is a binary data file that contains a bit image to be downloaded to a FPGA device. Data2BRAM can directly replace the Block RAM data in .bit files without the intervention of any Xilinx implementation tools. Hence, Data2BRAM both inputs and outputs .bit files. However, Data2BRAM can only modify existing .bit files. A .bit file is initially generated by the Xilinx implementation tools. Please refer to Xilinx implementation tools documentation for the details on creating .bit files. Since .bit files are binary data, they are not directly editable. Data2BRAM also provides some facilities for examining the content of .bit files. See section "Command Line Tool Usage" for usage details.

Verilog (.v) files

A .v file (Verilog) is a simple text file Data2BRAM outputs, that contains "defparm" records to initialize Block RAMs. This file is used primarily for pre- and post-synthesis simulation. Since a .v file is a simple text file, it is directly editable. However, since this file is a generated file, editing is not advised. Data2BRAM allows the free-form use of both "//" and "/*...*/" commenting styles. See sections "Command Line Tool Usage" for usage details.

VHDL (.vhd) files

A .vhd file (VHDL) is a simple text file Data2BRAM outputs, that contains "bit_vector" constants to initialize Block RAMs. These constants can then be used in "generic maps" to instance an initialized Block RAM. This file is used primarily for pre- and post-synthesis simulation. Since a .vhd file is a simple text file, it is directly editable. However, since this file is a generated file, editing is not advised. Data2BRAM allows the free-form use of both "//" and "/*...*/" commenting styles. See sections "Command Line Tool Usage" for usage details.

UCF (.ucf) files

A .ucf file (User Constraints File) is a simple text file Data2BRAM outputs, that contains "INST" records to initialize Block RAMs. Since a .ucf file is a simple text file, it is directly editable. However, since this file is a generated file, editing is not advised. Data2BRAM 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.

Usage Overview

Figure 2 portrays simplified tool flow views for CPU software and FPGA design. Some minor steps are left out of the diagrams for clarity.

On the left side of Figure 2, 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 also 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 it can be programmed into the target' s boot flash. Alternatively, the executable portion of a .drf file can be downloaded to a target via a symbolic debugger, and the debug portion can be used to symbolically debug the executable code image.

On the right side of Figure 2, FPGA source code is used in the form of .v, .vhd, and .edn files. These files are either used in various styles of hardware simulation or are synthesized into .edn intermediate files. A .ucf (user constraints file) and the intermediate .edn file are then run through NGDBuild, MAP and PAR to produce an .ncd file. Bitgen then converts the .ncd file into an FPGA .bit file that can be used to configure the FPGA. The .bit file can be either downloaded to the FPGA directly, or programmed into the FPGA' s boot configure flash.

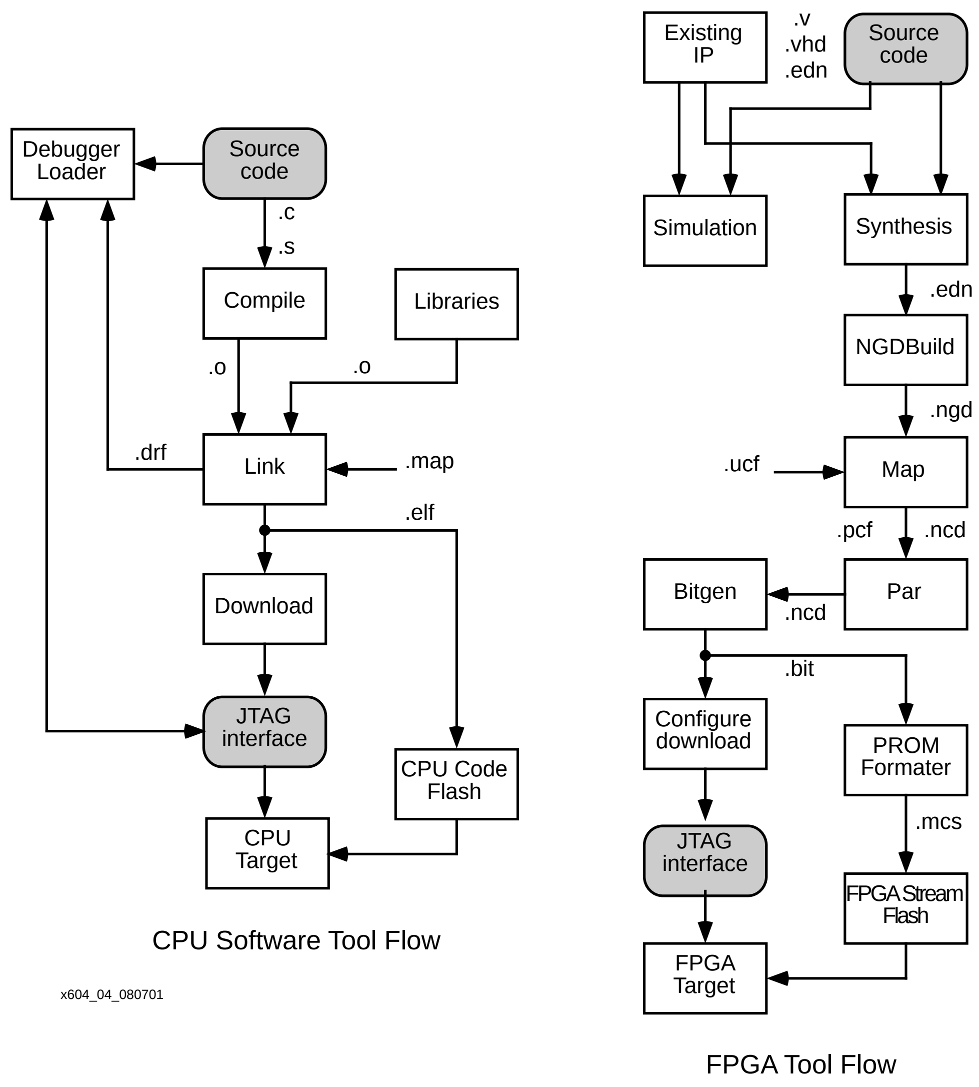

Figure 2 Simplified SW and HW Tool Flows

Although simplified, Figure 2 give an accurate representation of how the two tool flows operate within discrete-chip CPU/FPGA designs: two separate source bases, two separate bit images, and two separate boot mechanisms.

When integrating a discrete-chip CPU/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/FPGA bit images. Also, the tight integration of CPU and FPGA requires a much closer coupling of the FPGA simulation process. To produce combined bit images, Data2BRAM combines the CPU/FPGA tool flow outputs, while leaving the two flows themselves unchanged.

There are three distinct Data2BRAM tool uses.

1. For software designers that utilizes Data2BRAM as a command line tool to generate updated a .bit files. Refer to sections "Command Line Tool Usage" for usage details.
2. For hardware designers that integrates Data2BRAM with the Xilinx implementation tools. Refer to sections "Integrated Xilinx Implementation Tool Usage" for usage details.
3. Utilizing Data2BRAM as a command line tool to generate behavioral simulation files.

Process Overview

This section provides an overview of the data flow through Data2BRAM, and summarizes the design factors necessary when mapping CPU software code to a BRAM implemented address spaces.

This overview represents only a logical layout and grouping. FPGA logic must be constructed to translate CPU address requests into physical BRAM selection. The design of that FPGA logic is beyond the scope of this document.

Following are the design considerations for BRAM-implemented address spaces:

- BRAMs come in fixed-size widths and depths, and CPU address spaces might need to be much larger in width and depth than a single BRAM. Hence, multiple BRAMs need to 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 BRAM to obtain that data. Hence, byte-linear CPU data must be interleaved by the bit width of each BRAM and by the number of BRAMs in a single bus access. However, the relationship of CPU addresses to BRAM locations must be regular and easily calculable.
- CPU data must be located in a BRAM-constructed memory space relative to the CPU linear addressing scheme, not to the logical grouping of multiple BRAMs.
- 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 BRAM port sizes. 1, 2, 4, 8, and 16 bits for Virtex and Virtex-E devices and 1, 2, 4, 8, 16, and 32 bits for Virtex-II and Virtex-II Pro devices. Refer to Table 1 and Table 2.
- Addressing must account for the differences in instruction and data memory space. Since 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 BRAMs affect the access time. Evaluate the access time after implementation to verify that it meets the design specifications.

Given these considerations, refer to the diagram in Figure 3. The diagram graphically represents a 16 Kbyte address space from CPU address 0xFFFFC000 to 0xFFFFFFFF, constructed from the logical grouping of thirty-two 4 Kbit BRAMs. Each BRAM is configured to be 8 bits wide, and 512 bytes deep. CPU bus accesses are 8 BRAMs (64 bits) wide, with each column of BRAMs occupying an 8 bit wide slice of a CPU bus access called a "Bit Lane." Each row of 8 BRAMs 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 BRAMs is grouped together into a contiguous address space called an "Address Block."

Note Virtex, Virtex-E, Spartan2, and Spartan2E use 4 Kbit BRAMs. Virtex-II and Virtex-II Pro use 16 Kbit BRAM.

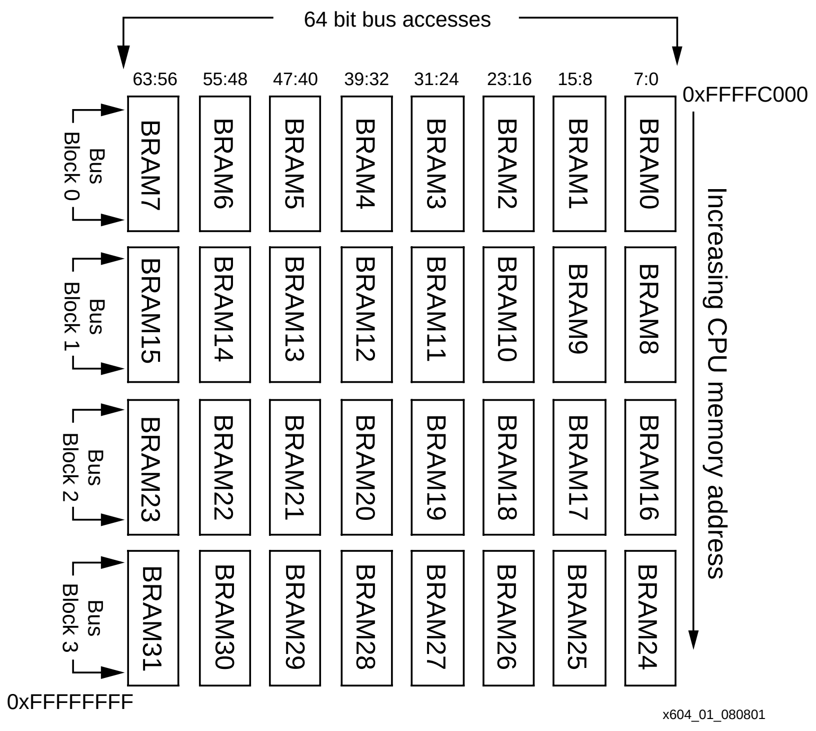

Figure 3 Example BRAM Address Space Layout

The address space, or Address Block, shown in Figure 3 consists of four Bus Blocks. The upper right corner address is 0xFFFFC000 and the lower left-hand corner address is 0xFFFFFFFF. Because a bus access obtains 8 data bytes across eight BRAMs, byte-linear CPU data must be “interleaved” by 8 bytes in the BRAMs. In this example, byte 0 goes into the first byte location of Bit Lane BRAM7; byte 1 goes into the first byte location of Bit Lane BRAM6; and so forth, to byte 7. However, CPU data byte 8 goes into the second byte location of Bit Lane BRAM7; byte 9 goes into the second byte location of Bit Lane BRAM6 and so forth, repeating until CPU data byte 15. This interleave pattern repeats until every BRAM 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 will be seen in section "BMM (Block RAM Memory Mapping) 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 referred to as byte lane mapping or more accurately, bit lane mapping, because these formulas are not restricted to byte-wide data. This is similar to the process embedded software programmers used when programmed CPU code is placed into the banks of fixed-size EPROM devices.

As mentioned previously, byte lane mapping is similar to a process already used by embedded CPU software programmers. However, byte lane mapping BRAMs differs in some important ways as the following describes:

- 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. Hence, little or no configuration options are provided. By contrast, the number and organization of FPGA BRAMs are completely "soft" (within FPGA limits), and any tool for byte lane mapping for BRAMs must support a vast set of device arrangements.
- Existing byte lane mapping tools assume some kind of ascending order of the physical addressing of byte-wide devices, because board-level hardware is built that way. By contrast, FPGA BRAMs have no fixed usage constraints and can be grouped together with BRAMS anywhere within the FPGA fabric. For clarity in these examples, Figure 3 displays BRAMs in ascending order. However, BRAMs can be configured in any order.
- Discreet storage devices are almost always only one or two bytes (8 or 16 bits) wide, or rarely, 4 bits wide. Existing tools usually assume that all storage devices have a single width. Virtex BRAM, however, can be configured in several widths, depending on the needs of the hardware designer, Table 1 and Table 2 specify the Virtex and Virtex-II BRAM widths.
- Existing tools have limited configuration needs so that a simple command line interface will suffice. BRAM usage adds more

complexity and warrants a human-readable syntax to describe the mapping between address spaces and BRAM utilization.

Table 1 Virtex, Virtex-E, and Spartan-IIe BRAM configurations

Component	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 2 Virtex-II and Virtex-II Pro BRAM configurations

Component	Data Cells		Parity Cells Currently Unused	
	Data Depth	Data Width	Depth	Width
RAMB16_S1	16384	1	-	-
RAMB16_S2	8192	2	-	-
RAMB16_S4	4096	4	-	-
RAMB16_S9	2048	8	-	-
RAMB16_S18	1024	16	-	-
RAMB16_S36	512	32	-	-

Note Officially, Virtex-II and Virtex-II Pro parts contain 18 kbit BRAMs; 16 Kbit of data, and 2 Kbits of parity. Data2BRAM currently does not support the use of the parity bits and, therefore, these BRAMs are referred to as 16 Kbit BRAMs.

BMM (Block RAM Memory Mapping) File Syntax

This section provides details of the syntax used in (.bmm) files.

Listing 1 shows the text-based syntax created to describe the organization of BRAM usage in a flexible and readable form. The address space defined by Listing 1 is the same BMM definition the address shown graphically in Figure 3.

BMM is oriented towards human readability and is similar 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 BRAMs.
- In-file documentation. As with any high-level computer language, allowing plain-text documentation embedded within the file contents preserves knowledge and promotes understanding. 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 the user to specify the data transposition in semi-graphical terms, while alleviating the need to specify the exact details of the address-to-BRAM algorithm. The computer then infers the algorithm details for the desired mapping.

For those with a software background, the Backus-Naur Form (BNF) specification for BMM syntax is found in Listing 3. Please be aware of the following important notational details:

- Keywords are shown here in uppercase, but are actually case-insensitive.
- Listing 1 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 also ignored. As many items as desired can appear on a single line.
- Comments can be either of two types:

1. `/*…*/` brackets a comment block of characters, words, or lines. This style of comments can be nested.
2. `//` means everything to the end of the current line is treated as a comment.

- Numbers can be entered as decimal or hex. Hex numbers use the “`0xXXX`” notation form.

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

```
ADDRESS_BLOCK ram_cntlr RAMB4 [start_addr:end_addr]
    .
    .
END_ADDRESS_BLOCK;
```

An `ADDRESS_BLOCK` and `END_ADDRESS_BLOCK` block keywords define a single contiguous address space. The mandatory name following the `ADDRESS_BLOCK` 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 Figure 3. A BMM file can contain multiple `ADDRESS_BLOCK` definitions, even for the same address space, as long as the name for each `ADDRESS_BLOCK` is unique.

Following the address space name is a keyword that defines what type of memory device the `ADDRESS_BLOCK` will be constructed. Currently, there are three device types defined.

1. RAMB4
2. RAMB16
3. MEMORY

The RAMB4 keyword defines the memory device as a 4-Kbit BRAM found in Virtex and Virtex-E parts. RAMB16 defines the memory device as a 16-Kbit BRAM found in Virtex--II and Virtex-II Pro parts. The correct keyword must be used for the FPGA part 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_BLOCK.`

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, Data2BRAM assumes that the smaller of the two values is the `start_addr`, and the larger is the `end_addr`.

Inside an `ADDRESS_BLOCK` 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 BRAM Bit Lane definitions that are accessed by a parallel CPU bus access. In the case of Listing 1, there are four, which correspond to the four Bus Block rows in Figure 3.

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 at the first, and highest addressed Bus Block is defined last. In the case of Listing 1, the first Bus Block would occupy CPU addresses 0xFFFFC000 to 0xFFFFCFFF. This is the same as the first row of BRAMs in Figure 3. The second Bus Block would occupy CPU addresses 0xFFFFD000 to 0xFFFFDFFF, repersents the second row of BRAMs in Figure 3. This pattern repeats in accending order until the last Bus Block.

Note The top-to-bottom order in which Bus Blocks are defined, also controls the order in which Data2BRAM will fill them with data; top first, up to the bottom.

A Bit Lane definition selects which bits out of a CPU bus access are assigned to which BRAMs. Each definition takes the form of a BRAM instance name followed by the bit numbers the Bit Lane occupies. The instance name must be preceded by the hierarchy path of the BRAM as used in the HDL design. The syntax is as follows:

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

Note Normally the bit numbers are give as shown in the order above, `[MSB_bit_num:LSB_bit_num]`. If the order is reversed to have the LSB first, and the MSB second, Data2BRAM will bit-reverse the Bit Lane value before placing it into the BRAM.

Just 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 infered to be the most significant Bit Lane value, and last defined is the least significant Bit Lane value. In the case of Figure 3. the most significant Bit Lane is BRAM7, and least significant Bit Lane is BRAM0. As seen in Listing 1, this corresponds with the order in which the Bit Lanes are defined.

It is also important to understand how Data2BRAM 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 was 0xB47DDE02826A8419, then the value 0xB4 would be the first value to be set into a BRAM.

Given the Bit Lane order, BRAM7 would be set to 0xB4, BRAM6 to 0x7D, etc. 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 the input data is exhausted. Figure 4 expands the first Bus Block of Figure 3 to graphically illustrate this process.

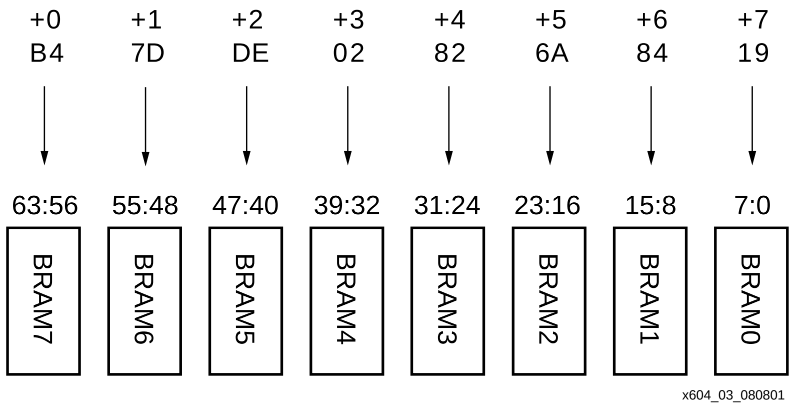

Figure 4 Bit Lane Fill Order

Note The Bit lane definitions must match the hardware configuration. If the BMM is defined different 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" BRAM devices, the physical row/column location within the FPGA can be indicated. The following are examples of the physical row/column location:

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

or

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

The `LOC` keyword is used by the designer to hard locate the corresponding BRAM. 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 above example indicates that the Implementation tools located the corresponding BRAM at row 3 and column 5, during BIT file generation (see section "Integrated Xilinx Implementation Tool Usage" for more detail on back annotated BMM files). These definitions are inserted after the bus bit values, and the terminating semicolon. These location constraints will also override any existing constraints in an .ucf file.

Note The RxCx syntax is used with the "RAMB4" keyword, for example, Virtex and Virtex-E parts. The XxYx syntax is used with the "RAMB16" keyword for Virtex-II and Virtex-II Pro parts.

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 .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 can be seen in Listing 1, .mem files will be created for all of the BRAMs in the second Bus Block.

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

- While the examples in this document use only byte wide for clarity, the same principles apply to whatever data width a BRAM is configured for.
- 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 BRAMs in a Bus Block must equal the range of addresses infered by a Bus Block's start and end addresses.
- All Bus Blocks must be the same number of bytes in size.
- A BRAM instance name can only be specified 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.

Data2BRAM checks for all of these conditions and emits an error message if a violation is detected.

Command Line Tool Usage

Command line functionality falls into several categories. The following sections descibes those categories, and shows typical uses.

Block RAM Memory Map File Syntax Checking

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

```
data2bram -bm my.bmm
```

Data2BRAM 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 the designer 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 ELF or MEM data files into a different format. Principally, data files are converted into BRAM initialization files for Verilog and VHDL, or UCF BRAM initialization records. A conversion to all three formats would be invoked as:

```
data2bram -bm my.bmm -bd code.elf -o uvh output
```

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

```
data2bram -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.

Lastly, 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 for the device that occupies a BitLane. This is true for both BRAM and external memory devices. With a command line invoked as:

```
data2bram -bm my.bmm -bd code.elf -o m output
```

and a BitLane appeared as:

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

the MEM file “ram0.mem” would be produced that contained 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 BitLane 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 will be confined to only those that set of Adress Blocks. A -bm option might be modified as:

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

In this way, data translation will only take place to the Address Blocks "mem1" and "mem2"; even if data in "code.elf" matches another Address Block. This allows the designer to *steer* differnet 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. See the -i option for more information.

BIT File Block RAM Replacement

The Data2BRAM provides the facility to iterate new BRAM data into a BIT file without the need to rerun the Xilinx implementation tools. In conjunction with new ELF and a BMM file, Data2BRAM updates the BRAM initialization in a BIT file image, and and outputs a new BIT file. Additionally, tag filtering can be include. This facility would be invoked as:

```
data2bram -bm my.bmm -bd code.elf -bt my.bit -o b new.bit
```

This would produce a new BIT file called "new.bit", with the proper BRAM contents replaced with the contents of "code.elf".

Note For proper operation, the BMM file *must* have "LOC" or "PLACED" constraints for each BRAM. These constraints can be added by hand, but is most often obtained as a annotated BMM file from Bitgen. See the "Integrated Xilinx Implementation Tool Usage" section for more infomation.

The process yeilds a new BIT file significantly faster than rerunning the implementation tools (from 100 to 1000 times). This facility is meant primarily as a means to include new CPU software code into a design when the logic portion of the design isn't changing. Most othen this will be used by a software developer that no access (nor understanding) of the Xilinx implementation tools.

Examining BIT and ELF File Contents

Data2BRAM provides the ability to examine, or *dump*, the contents of ELF and BIT data files. The dump content isin a text hex 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 changes some input data file dependent information is displayed.

ElF dumps are invoked as:

```
data2bram -bd code.elf -d
```

This dump will show the contents of each section within the ELF file. Using the "e" option will diplay additional information about each section. Using the "r" option will include some redundant ELF header information.

Note ELF files contain much more data than what is used for Data2BRAM data translation (symbols, debug, etc.). Only those sections labeled "Program header record" are considered by Data2BRAM for data translation.

BIT dumps are invoked as:

```
data2bram -bm my.bmm -bt my.bit -d
```

Each bit stream command is decoded and displayed. Those commands that contain bit field flags will have each bit field described. Commands that contain non-BRAM data chunks will be displayed as plain hex dumps. Since BRAM data is encoded within bit streams, Data2BRAM will display BRAM data as decoded hex dumps.

These dumps are used primarily for debugging purposes. However, they are also useful for comparing binary ELF and BIT files with simple, and human friendly, text tools.

Miscellaneous Functionality

Data2BRAM has twoo other options to control it's behavior.

The -i option will tell Data2BRAM to ignore any data in an ELF or MEM file that is outside of any Address Block within the BMM file. This allows data files to be used that have much more data in them the BMM file knows about. For example, a master design code file, of which only a small portion is destined for BRAM memories.

The -u option will force Data2BRAM to output text output files for all Address Spaces, even if no data was transformed into an Address Space. Depending on file type, an output file will be either empty, or will contain initializations of all zero. If this option is not used, only Address Spaces that receive transformed data will be output.

Integrated Xilinx Implementation Tool Usage

This section describes how Data2BRAM functionality is integrated into the Xilinx Implementation Tool Flow. This flow allows the hardware designer to associate Block RAM with a BMM file directly from within Xilinx Implementation Tools. Access to Data2BRAM functionality is done by utilizing a sub-set of Data2BRAM options in NGDBuild, BitGen, NGDAnno, and FPGA Editor. Figure 5 illustrates the software flow and the file dependencies. The following sections describe each option and its effect in the software flow.

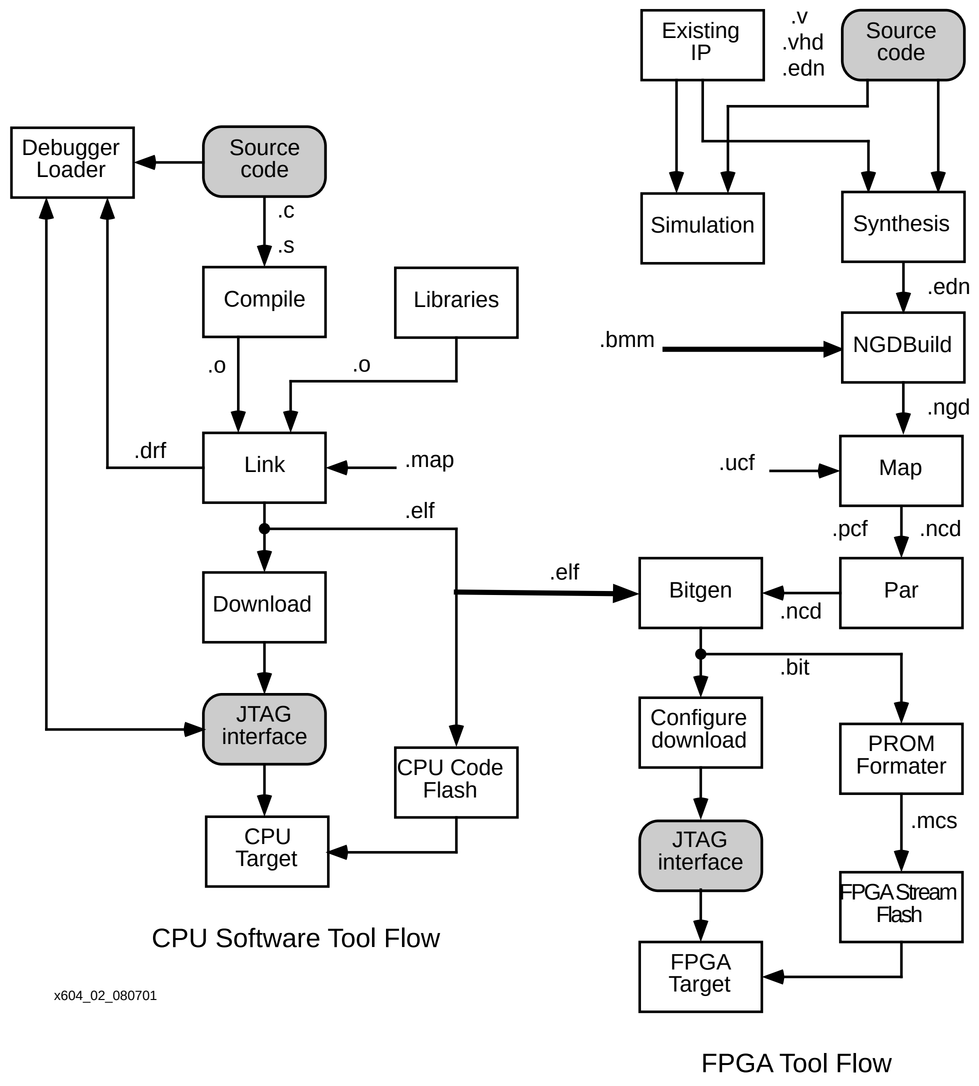

Figure 5 Integrated Data2BRAM and Implementation Tool Flow

NGDBuild Usage

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

Usage: This option allows you to specify the name and the path of the BMM file. If the BMM file is the same name as the design name and is located in the same directory as the design netlist, that 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. The Xilinx Design Manager/Flow Engine (DMFE) does not track changes to the BMM file. In this case, the user is responsible to restart the Xilinx software flow themselves.

Functionality: NGDBuild creates the `BMM_FILE` property in the 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 BRAMs 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 Data2BRAM). Also, any BRAM placement constraints that appear in the BMM file are applied to the corresponding BRAM.

Note The -bm switch is supported in ISE. Design Manager users are required to use the *Template Manager and Customize* to set this option.

MAP and PAR Usage

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

BitGen Usage

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

Usage: The -bd switch specifies the path and file name of the .elf file used to populate the BRAMs specified in the BMM file. The address information contained in the .elf file allows Data2BRAM to determine which `ADDRESS_BLOCK` to place the data.

Functionality: BitGen passes the -bd switch with the `<filename>` and any tag information to Data2BRAM. Data2BRAM processes the BMM file specified during the NGDBuild phase and the ELF file is used to internally create the BRAM initialization strings for each BMM defined BRAM. The initialization strings are then used to update the .ncd file before the .bit file is created.

Placement information of each BRAM is provided by the .ncd file. Any BRAM placement constraints that appeared in the BMM file are already reflected in the .ncd information. All other BRAMs are assigned placement constraints by previous tool steps. These placement constraints are passed to Data2BRAM 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 BRAMs will just have zero'ed content.

In addition to the original BMM file contents, this file contains the placement information for all of the BRAMs 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 Data2BRAM.

Note The -bm switch is supported in ISE. Design Manager users are required to use the *Template Manager and Customize* to set this option.

NGDAnno Usage

```
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 BRAMs specified in the BMM file. The address information contained in the .elf file allows Data2BRAM to determine which `ADDRESS_BLOCK` to place the data.

Functionality: NGDAnno passes the -bd switch with the `<elf_filename>` to Data2BRAM. Data2BRAM processes the BMM file specified during NGDBuild and the .elf file is used to create the BRAM initialization strings for each of the constrained BRAMs. The initialization strings are then used to update the .ncd file before the .bit file is created.

Placement information of the BRAM is provided from the .ncd file to Data2BRAM. 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 Data2BRAM.

Note The -bm switch is supported in ISE. Design Manager users are required to use the *Template Manager and Customize* to set this option.

FPGA Editor Usage

```
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 BRAMs specified in the BMM file. The address information contained in the ELF file allows Data2BRAM to determine which ADDRESS_BLOCK to place the data.

Functionality: BRAMs specified in the BMM file will be marked READ-ONLY in FPGA Editor. Any changes made in FPGA Editor to the contents of the BRAM mapped in the BMM file will not be retained even if the user writes out the NCD. The ELF file contents must be changed to permanently change the contents of the BRAMs.

Note The -bm switch is supported in ISE. Design Manager users are required to use the *Template Manager and Customize* to set this option.

Limitations

The following section describes the limitations of Integrated implementation tool usage of Data2BRAM functionality:

- Tools: XDL and NCDRead do not call Data2BRAM to update the BRAM initialization strings. This results in different values than those seen in FPGA Editor, BitGen, or NGDAnno.
- BRAMs specified in the BMM file may be trimmed during MAP if connected incorrectly. This may result in an error when Data2BRAM is run.
- Translating CPU addresses to physical BRAM addresses must be done as part of the HDL hardware design.

Command Line Option Reference

Data2BRAM has fairly simple command line options. The overall syntax is as follows:

```
data2bram
 <-bm filename [.bmm]> |
 <<[-bm filename [.bmm]]>
  <-bd filename [<.elf>|<.mem>] [<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>] [<tag TagName <TagName>...>]>...
  <-bt filename [.bit]> <-o b filename [.bit]>> |
 <<-bm filename [.bmm]> <-bt filename [.bit]> -d>> |
<-f filename [.opt]> |
 <-q [e|w|i]> |
 -u |
 -h
```

Table 3 Command Line Options

Option	Description
-bm *filename*	Name of the input BMM file. If the file extension is missing, a .bmm file extension is assumed. If this is option not unspecified, the ELF or MEM root file name with a .bmm extension is assumed. If only this option is given, then the BMM file is merely syntax checked and any errors are reported. Only one -bm option can be used.
-bd *filename*	Name of the an 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 will be used for translation. All other input file data outside of the TagName address spaces will be ignored. If no further options are specified, "-o u filename" functionality is assumed. One or more -bd options can be used.

Table 3 Command Line Options

Option	Description
-bt *filename*	Name of the input BIT file. If the file extension is missing, .bit is assumed. If the -o option is not specified, the output BIT file name will have the same root file name as the input BIT file, with a "_rp" appended to the end. A .bit file extension is assumed. Otherwise, the output BIT file name will be as specified in the -o option. Also, the device type is automatically set from the BIT file header, and the -p option will have no effect.
-o u | v | h | m | b *filename*	The name of the output file(s). The string preceding the file name indicates which file formats are to be output. No spaces can separate the file type characters, but can appear in any order. As many, or as few file type characters can be used at once. The file type characters mean: ' 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. ' b' = BIT file format, a .bit file extension. The *filename* applies to all specified output file types. If the file extension is missing, the appropriate file extension will be added to specified output file types. If the file extension is specified, the appropriate file extension will be added to the remaining file formats. An output file contains data from all translated input data files.
-u	Update -o text output files for all Address Spaces, even if no data was transformed into an Address Space. Depending on file type, an output file will be either empty, or will contain initializations of all zero. If this option is not used, only Address Spaces that receive transformed data will be output.
-p *partname*	Name of the target Virtex part. If this is unspecified, a ' xcv50' part is assumed. Use the -h option to obtain the full supported part name list.

Table 3 Command Line Options

Option	Description
`-i`	Ignore ELF or MEM data that is outside the address space defined in the BMM file. Otherwise, an error will be generated.
-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 BRAM. 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 can be used at once. These modifiers mean: ' e' = EXTENDED mode. Display additional information for each ELF section. ' r' = RAW mode. This includes some redundant ELF information.
-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, just in a text file instead. A option, and its items, must appear on the same text line. However, as many switchs can appear on the same text line as desired. This option can be used only once, and a .opt file can' t contain a -f option.
-q e|w|i	Disable the output of Data2BRAM messages. The string following the option indicates which messages types will be disabled. No spaces can separate the message type characters, but can appear in any order. As many, or as few message type characters can be used at once. The message type string is optional. Leaving the message type blank is equivalent to using "-q wi". The message type characters mean: ' e' = Disable ERROR messages. ' w' = Disable WARNING messages. ' i' = Disable INFO messages.
-h	Print help text, plus supported part name list.

Listing 1- Example Address Space Map 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_BLOCK 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_BLOCK;
```

Listing 2- Modified Backus-Naur Form (BMM) Syntax

```
Address_block_keyword       ::=   “ADDRESS_BLOCK”;

End_address_block_keyword   ::=   “END_ADDRESS_BLOCK”;

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” |
                                  “MEMORY”;

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“;” ;
```