Data2MEM ユーザー ガイド

UG658 (v12.2) 2010 年 7 月 23 日

XILINX®

本資料は英語版 (v.12.2) を翻訳したもので、内容に相違が生じる場合には原文を優先します。
資料によっては英語版の更新に対応していないものがあります。
日本語版は参考用としてご使用の上、最新情報につきましては、必ず最新英語版をご参照ください。

目次

XILINX®

第 1 章

Data2MEM の概要

『Data2MEM ユーザー ガイド』では、Data2MEM ソフトウェア ツールで、ザイリンクス FPGA ファミリのブロック RAM メモリの内容設定がどのように自動化および簡素化されるか説明します。

次のセクションが含まれています。

- Data2MEM の概要
- Data2MEM の機能
- Data2MEM の使用
- CPU ソフトウェアと FPGA デザインのツール フロー
- Data2MEM ブロック RAM がインプリメントされたアドレス スペースについての注意事項
- Spartan®-3、Spartan-3A、Spartan-3E デバイスのブロック RAM コンフィギュレーション

Data2MEM の概要

Data2MEM は、複数のブロック RAM (1 つの連続した論理アドレス スペースで構成される) で使用される連続したデータ ブロックのデータを変換するツールです。Virtex® シリーズ デバイスとシングル チップ上のエンベデッド CPU を使用する場合に Data2MEM を使用すると、CPU のソフトウェア イメージを FPGA ビットストリームに含めることができます。この結果、CPU ソフトウェアが FPGA ビットストリーム内のブロック RAM のメモリから実行できるようになり、効率的で柔軟な方法で CPU ソフトウェアのパーツを FPGA デザイン ツール フローに統合できます。また、Data2MEM を使用することで、CPU を含まないデザインのブロック RAM の初期化も簡素化できます。

メモ： Data2MEM は、暗号化または圧縮オプションを使用せずに作成されたビット ファイルをアップデートするためだけに使用できます。

Data2MEM は、プロセスをシンプルな手法に自動的に変更するだけでなく、次も可能にします。

- FPGA および CPU ソフトウェア設計両方の既存ツール フローへの影響が最小限
- 1 つのツール フローで発生した時間的な遅れがほかのツールでのテストや問題修正に与える影響を制限
- プロセスを分けて、ステップ数を最小限に抑制
- 1 つのツール フローを使用するユーザー (CPU ソフトウェア または FPGA 設計者) がその他ツール フローのステップやその詳細を学ぶ必要性を削減

Data2MEM は、次の OS で使用できます。

- Linux
- Windows XP
- Windows Vista

Data2MEM の機能

Data2MEM では、次のデバイスがサポートされます。

- Virtex®-4
- Virtex-5
- Virtex-6
- Spartan®-3A
- Spartan-3AN
- Spartan-3A DSP
- Spartan-6

Data2MEM には、次の機能があります。

- ブロック RAM の使用方法やワード数を記述したテキスト形式構文を含む新規の BMM (ブロック RAM メモリ マップ) ファイルを読み込みます。この構文には、CPU バス幅とビット レーン インターリーブなどの情報も含まれます。
- ブロック RAM モデルから使用可能な複数のデータ幅に対応します。
- ELF (Executable and Linkable Format) ファイルを CPU ソフトウェア コード イメージの入力として使用します。サードパーティの CPU ソフトウェア ツールを使用する場合に、そのファイル形式の CPU ソフトウェア コードを変換する必要はありません。
- MEM 形式のテキスト ファイルをブロック RAM の入力ファイルとして読み込みます。このテキスト形式ファイルは、手書きで作成できるだけでなく、マシンで生成することもできます。
- オプションで BIT および ELF ファイルの内容からフォーマットされたテキストをダンプします。
- 合成前後のシミュレーション用の初期化のために Verilog および VHDL ファイルを生成します。
- 初期化データを配置配線 (PAR) 後のシミュレーションに統合します。
- サードパーティのメモリ モデルを使用して Verilog シミュレーション用に MEM ファイルを生成します。
- その他のザイリンクス インプリメンテーション ツールを使用せずに BIT ファイルのブロック RAM の内容を直接置換できるので、インプリメンテーション時間が短縮されます。
- コマンド ライン ツールまたはザイリンクス ツール フローに統合された一部分として起動できます。
- Windows や Linux などのよくあるテキスト ライン終了タイプを認識し、同様に使用します。
- テキスト入力ファイルで構文をコメントにする場合、// および /*...*/ を使用できます。

メモ : Data2MEM は、暗号化または圧縮オプションを使用せずに作成されたビット ファイルをアップデートするためだけに使用できます。

Data2MEM の使用

Data2MEM は、次のプロセスに使用できます。

1. ソフトウェア デザインでコマンド ライン ツールとして使用し、アップデートされた BIT ファイルを生成します。詳細は、「コマンド ラインの使用」を参照してください。

   メモ : Data2MEM は、暗号化または圧縮オプションを使用せずに作成されたビット ファイルをアップデートするためだけに使用できます。

2. ハードウェア デザインで Data2MEM とザイリンクス インプリメンテーション ツールを統合します。詳細は、「ISE® Design Suite インプリメンテーション ツールの使用」を参照してください。

3. コマンド ライン ツールとして使用し、ビヘイビア シミュレーション ファイルを生成します。

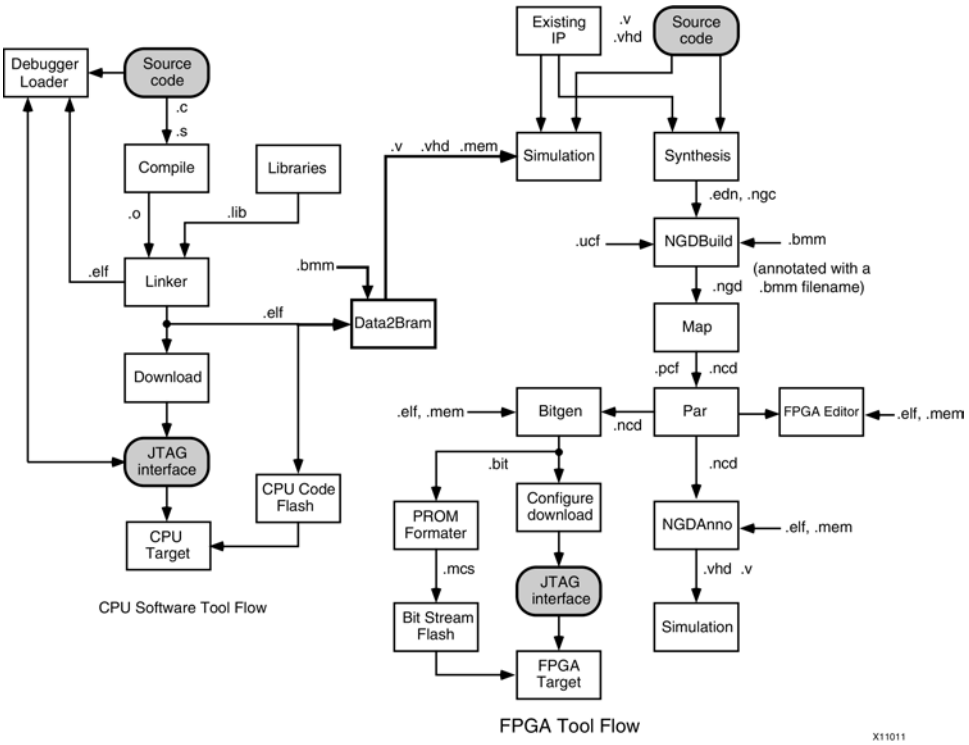
次の図には、ソース ベース 2 つ、ビット イメージ 2 つ、ブート フラッシュ デバイス 2 つが含まれており、CPU と FPGA デザインそれぞれのツール フローを正確に表しています。

別のチップにある CPU および FPGA デザインを 1 つの FPGA チップにまとめる場合、ソース ベースは別々にしておくことができます。つまり、ソースで処理されるツール フローの部分も別にしておくことができます。

ただし、FPGA チップを 1 つにすると、ブート イメージが 1 つになるので、CPU と FPGA のビット イメージを結合したものを含める必要があります。また、CPU と FPGA の統合を強固にするには、FPGA シミュレーション プロセス内で連結させる必要があります。Data2MEM は CPU と FPGA ツール フローを変更せずに 2 つの出力をまとめて、1 つのビット イメージを生成します。

次の図は、CPU ソフトウェアと FPGA デザインのツール フローを表しています。

ソフトウェアおよびハードウェアのツール フロー



次のセクションでは、CPU ソフトウェアのソース コードと FPGA デザインのデータ フローについて説明します。

CPU ソフトウェア ソース コードおよび FPGA ソース コード

次のセクションでは、CPU ソフトウェアのソース コードと FPGA デザイン コードが Data2MEM でどのように使用されるかについて記述します。

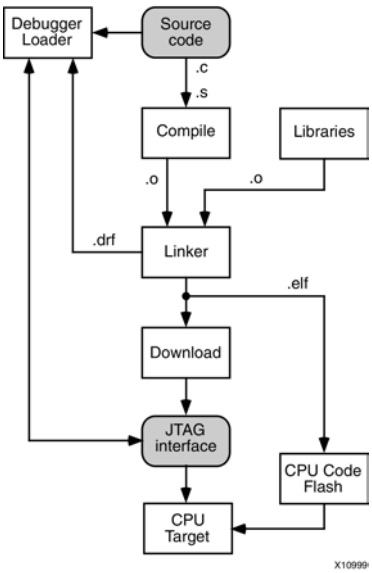
- CPU ソフトウェア ソース コード
- FPGA ソース コード

CPU ソフトウェア ソース コード

CPU ソフトウェア ソース コードには、高度な C ファイルおよびアセンブリ レベルの S ファイルが使用されます。これらのファイルは、object (.o) リンク ファイルにコンパイルされます。このオブジェクト ファイルとあらかじめ組み込まれているオブジェクト生成ライブラリが、単一の実行可能なコード イメージにまとめられます。

リンカの出力は、ELF ファイルです。ELF の内容は、JTAG を使用して Xilinx® Microprocessor Debugger (XMD) と共にダウンロードするか、不揮発性メモリに格納するか、小さい場合は BIT ファイルに格納します。

CPU Software Tool Flow

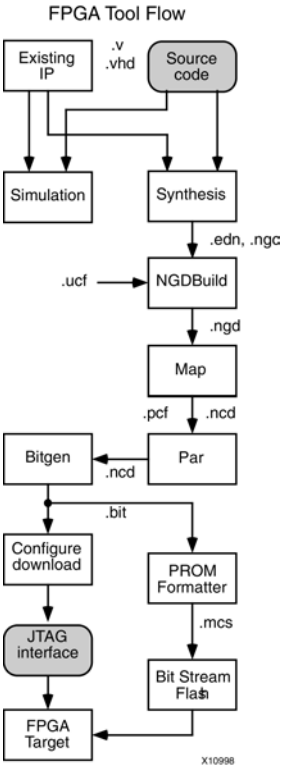


FPGA ソース コード

FPGA ソース コードのファイル形式は、Verilog (V)、VHDL (VHD) および Electronic Data Interchange Format (EDIF) ファイルになります。これらのファイルは次のデザイン フローで使用されます。

- さまざまなハードウェア シミュレーションで使用されるか、EDN または NGC 中間ファイルに合成されます。
- UCF (ユーザー制約ファイル) と EDN または NGC 中間ファイルが NGDBuild、MAP および PAR (配置配線) で実行され、Native Circuit Description (NCD) ファイルが作成されます。

  メモ : NGDBuild は、すべての入力デザイン ネットリストを変換し、その結果を 1 つのファイルに出力するプログラムです。
- この後、BitGen により、NCD ファイルが FPGA のコンフィギュレーションに使用可能な FPGA ビットストリーム (BIT) ファイルに変換されます。
- BIT ファイルは、FPGA に直接ダウンロードするか、FPGA のブート フラッシュにプログラムできます。



設計に関する考慮事項

このセクションでは、CPU ソフトウェア コードをブロック RAM がインプリメントされたアドレス スペースにマップする際の注意事項をまとめています。

次のフロー図は、ブロック RAM メモリのロジック レイアウトとグループ化について示しています。FPGA ロジックは、CPU アドレス リクエストを物理的なブロック RAM 選択に変換するように構築しなければなりませんが、このような FPGA ロジックのデザインについては、このマニュアルでは説明していません。

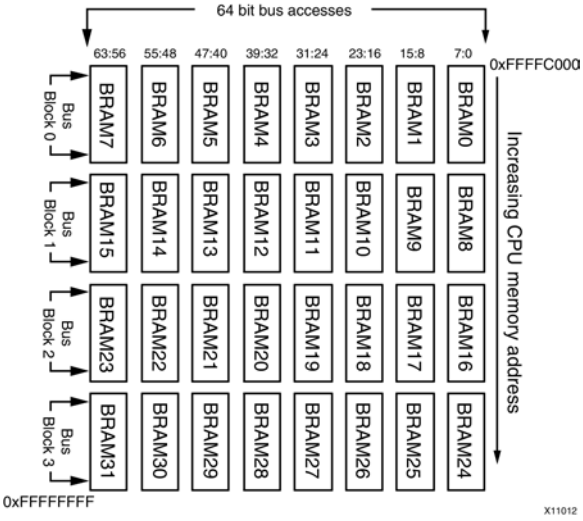
次は、ブロック RAM がインプリメントされたアドレス スペースについての注意事項です。

- ブロック RAM の幅とワード数は固定されているので、CPU アドレス空間に 1 つのブロック RAM よりも大きい幅とワード数が必要な場合があります。この場合は、複数のブロック RAM を論理的にまとめ、1 つの CPU アドレス スペースを作成する必要があります。
- 1 つの CPU バスのアクセスは、多くの場合、一度に 32 ビットまたは 64 ビット (4 または 8 バイト) など、複数バイトのデータ幅です。
- 複数バイトのデータによる CPU バスのアクセスでは、複数のブロック RAM にアクセスし、データを取得することもできます。このため、バイト リニアの CPU データは、各ブロック RAM のビット幅と単一バス アクセスのブロック RAM 数でインターリーブする必要があります。ただし、CPU アドレスとブロック RAM ロケーションは、規則的で容易に計算できるような関係にする必要があります。
- CPU データは、複数ブロック RAM の論理グループではなく、CPU リニア アドレス指定方法に関連した ブロック RAM 構成のメモリ空間にある必要があります。
- アドレス スペースは、連続している必要があり、また CPU バス幅の倍数である必要があります。バス ビット レーンのインターリーブは、Virtex® デバイスのブロック RAM ポート サイズでサポートされるサイズでのみ実行できます。デバイス ファミリ別のデータ バス サイズは次のようになります。
  - Spartan®-3 および Virtex-4 の場合は 1、2、4、8、9、16、18、32、36 ビット
  - Virtex-5 の場合は 1、2、4、9、18、36、72 ビット。詳細は、「デバイス ファミリ別ブロック RAM コンフィギュレーション」を参照してください。

    メモ : パリティを使用する場合は、Data2MEM ではパリティ ビットがデバイス データ バスの上位ビット (MSB) にあると認識されます。詳細は「ビット レーンの定義 (メモリ デバイスの使用)」を参照してください。
- アドレスの指定では、命令とデータのメモリ空間の違いを考慮する必要があります。命令空間は書き込み可能ではないため、アドレス幅に制限はありませんが、データ空間は書き込み可能なため、各バイトに書き込むことができるようにしておく必要があります。このため、各バス ビット レーンはアドレス指定可能にする必要があります。
- メモリ マップのサイズと各ブロック RAM のロケーションによって、アクセス時間は異なります。インプリメンテーション後のアクセス時間を評価し、デザイン仕様を満たしているかどうか検証してください。

  詳細は、「ソフトウェアおよびハードウェアのツール フロー」を参照してください。
- 論理的にグループ化された 32 個の 4 K ビット ブロック RAM で構成された CPU アドレスの 16 K バイト アドレス空間 (0xFFFFC000 ～ 0xFFFFFFFF) を表しています。
- 各ブロック RAM は 幅 8 ビット、ワード数 512 バイトになるようにコンフィギュレーションされています。
- CPU バス アクセスは 8 ブロック RAM (64 ビット) 幅で、ブロック RAM の各列は「ビット レーン」と呼ばれる 8 ビット幅スライスの CPU バス アクセスで占められます。
- バス アクセス内の 8 個のブロック RAM の各行は、「バス ブロック」にグループ分けされるので、各バス ブロックは幅 64 ビット、4096 バイトになります。
- ブロック RAM 全体は、「アドレス ブロック」という連続したアドレス スペースにまとめられます。

ブロック RAM のアドレス スペースのレイアウト例



上記のレイアウトのアドレス スペースには、4 つのバス ブロックが含まれています。右上端のアドレスは 0xFFFFC000 で、左下端のアドレスは 0xFFFFFFFF です。バス アドレスには 8 個のブロック RAM からの 8 データ バイトが含まれるため、バイト リニアの CPU データはブロック RAM の 8 バイトでインターリーブする必要があります。

この例では、左から右へ [0:7] や [8:15] のようにバイト インデックスが付いた 64 ビットのデータワードが使用されています。

- バイト 0 がブロック RAM7 ビット レーンの最初のバイト ロケーションに、バイト 1 がブロック RAM6 ビット レーンの最初のバイト ロケーションに入り、同様にバイト 7 まで挿入されます。
- CPU データ バイト 8 がブロック RAM7 ビット レーンの 2 つ目のバイト ロケーションに入り、バイト 9 がブロック RAM6 ビット レーンの 2 つ目のバイト ロケーションに入り、同様にバイト 15 まで繰り返されます。
- このようなインターリーブ パターンは、最初のバス ブロックの各ブロック RAM が満たされるまで繰り返されます。
- このプロセスは、メモリ スペース全体が満たされるか、または入力データがすべて挿入されるまで、バス ブロックごとに繰り返されます。

「BMM ファイルの構文」に記述されているように、ビット レーンおよびバス ブロックが定義されている順序が満たされる順序になります。この例の場合、ビット レーンは左から右へ、バス ブロックは上から下へ定義されています。

このプロセスは、バイト幅のデータのみに限られているわけではないので、「ビット レーン マップ」と呼ばれます。このプロセスは、プログラム済み CPU コードを固定サイズの EPROM デバイスのバンクに配置する際に使用する、埋め込み型ソフトウェアのプロセスと類似していますが、同一のものではありません。

この 2 つのプロセスの違いは、次のとおりです。

- CPU 埋め込み型システムの場合、数と構成が固定されたバイト幅の記憶デバイスをバイト レーン マップするため、カスタマイズされたソフトウェア ツールが使用されます。記憶デバイスの数と構成は変更できないため、このようなツールでは特定のデバイス配列のみが認識されます。そのため、コンフィギュレーション オプションはほとんどありません。FPGA ブロック RAM の数と構成は FPGA の制限内で変更可能なため、ブロック RAM のバイト レーン マップを実行するツールでは、さまざまなデバイス配列がサポートされている必要があります。
- 既存のバイト レーン マップ ツールでは、バイト幅デバイスの物理的アドレス指定を昇順で処理します。これは、ボード レベルのハードウェアが昇順で作成されているからです。FPGA ブロック RAM の場合は決まった使用規則がないため、FPGA チップ上のどこにあるブロック RAM でもグループ化できます。この例ではブロック RAM は昇順で表示されていますが、実際にはどの順にでもコンフィギュレーションできます。
- 記憶デバイスは、通常 1 バイトまたは 2 バイト (8 ビットまたは 16 ビット) 幅ですが、4 ビット幅もまれにあります。既存のツールでは、通常すべての記憶デバイスの幅が同じであるとして処理されますが、Virtex-4 および Virtex-5 のブロック RAM の場合は、ハードウェア要件に応じて、さまざまな幅にコンフィギュレーションできます。「デバイス ファミリ別ブロック RAM コンフィギュレーション」の表は、ブロック RAM の幅を示します。
- 既存ツールには、コンフィギュレーションでの要件に限りがあるため、シンプルなコマンド ライン インターフェイスで十分です。ブロック RAM を使用すると、複雑性が増すため、アドレス スペースとブロック RAM 間のマップを設計者が理解できるような構文が記述されます。

デバイス ファミリ別ブロック RAM コンフィギュレーション

このセクションでは、デバイス ファミリ別のブロック RAM コンフィギュレーションについて説明します。

- Spartan®-3、Spartan-3A、Spartan-3E デバイスのブロック RAM コンフィギュレーション
- Virtex®-4 デバイスのブロック RAM コンフィギュレーション
- 18 Kbit の Virtex-5 デバイスのブロック RAM コンフィギュレーション
- 36 Kbit の Virtex-5 デバイスのブロック RAM コンフィギュレーション

Spartan-3、Spartan-3A、Spartan-3E デバイスのブロック RAM コンフィギュレーション

プリミティブ	ワード数	データ幅	メモリ タイプ
RAMB16_S1	16384	1	RAMB16
RAMB16_S2	8192	2	RAMB16
RAMB16_S4	4096	4	RAMB16
RAMB16_S9	2048	8	RAMB16
RAMB16_S9	2048	9	RAMB18
RAMB16_S18	1024	16	RAMB16
RAMB16_S18	1024	18	RAMB18
RAMB16_S36	512	32	RAMB32
RAMB16_S36	512	36	RAMB36

Virtex-4 デバイスのブロック RAM コンフィギュレーション

プリミティブ	ワード数	データ幅	メモリ タイプ
RAMB16	16384	1	RAMB16
RAMB16	8192	2	RAMB16
RAMB16	4096	4	RAMB16
RAMB16	2048	8	RAMB16
RAMB16	2048	9	RAMB18
RAMB16	1024	16	RAMB16
RAMB16	1024	18	RAMB18
RAMB16	512	32	RAMB32
RAMB16	512	36	RAMB36

18 Kbit の Virtex-5 デバイスのブロック RAM コンフィギュレーション

プリミティブ	ワード数	データ幅	メモリ タイプ
RAMB18	16384	1	RAMB16
RAMB18	8192	2	RAMB16
RAMB18	4096	4	RAMB16
RAMB18	2048	8	RAMB16
RAMB18	2048	9	RAMB18
RAMB18	1024	16	RAMB16
RAMB18	1024	18	RAMB18
RAMB18SDP	512	32	RAMB32
RAMB18SDP	512	36	RAMB36

36 Kbit の Virtex-5 デバイスのブロック RAM コンフィギュレーション

プリミティブ	ワード数	データ幅	メモリ タイプ
RAMB36	32768	1	RAMB32
RAMB36	16384	2	RAMB32
RAMB36	8192	4	RAMB32
RAMB36	4096	8	RAMB32
RAMB36	4096	9	RAMB36
RAMB36	2048	16	RAMB32
RAMB36	2048	18	RAMB36
RAMB36	1024	32	RAMB32
RAMB36SDP	512	64	RAMB32

XILINX®

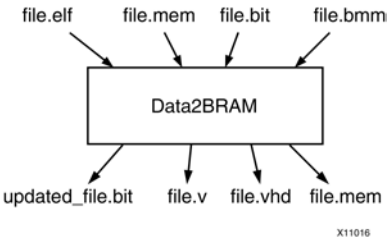
第 2 章

入力および出力ファイル

この章では、各入力および出力のファイル タイプとその使用方法または Data2MEM での作成方法について説明します。次のセクションが含まれています。

- ブロック RAM メモリ マップ (BMM) ファイル
- Executable and Linkable Format (ELF) ファイル
- メモリ (MEM) ファイル
- ビットストリーム (BIT) ファイル
- Verilog ファイル
- VHDL ファイル
- User Constraints File (UCF) ファイル

次の図は、Data2MEM で使用されるさまざまな入力および出力ファイルを示しています。



ブロック RAM メモリ マップ (BMM) ファイル

ブロック RAM メモリ マップ ファイル (BMM) は、各ブロック RAM がどのように連続した論理データ空間を構成するかが構文で記述されたテキスト ファイルです。Data2MEM は、このファイルを入力ファイルとして使用してデータを最適な初期化形式に変換します。

BMM ファイルは、次のように作成できます。

- 手動
- Data2MEM を使用して BMM ファイル テンプレートを生成
- 自動スクリプト

テンプレートは特定デザイン用にカスタマイズできます。

BMM ファイルはテキスト ファイルなので、直接編集できます。BMM ファイルでは、コメントに対して // と /*...*/ の両方とも使用できます。

詳細は、「BMM の機能」および「BMM ファイルの構文」を参照してください。

次の「BMM ファイルの例」は、ブロック RAM 構造を柔軟で読みやすいようにテキスト形式で記述した構文です。

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

ADDRESS_SPACE ram_cntlr RAMB16 [0xFFFF0000:0xFFFFFFFF]

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

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

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

  // Bus access map for next higher 16k, CPU address 0xFFFFC000 - 0xFFFFFFFF
  BUS_BLOCK
    top/ram_cntlr/ram31 [63:56];
```

```
    top/ram_cntlr/ram30 [55:48];
    top/ram_cntlr/ram29 [47:40];
    top/ram_cntlr/ram28 [39:32];
    top/ram_cntlr/ram27 [31:24];
    top/ram_cntlr/ram26 [23:16];
    top/ram_cntlr/ram25 [15:8];
    top/ram_cntlr/ram24 [7:0];
  END_BUS_BLOCK;

END_add_SPACE;
```

Executable and Linkable Format (ELF) ファイル

ELF ファイルは、バイナリ データ ファイルで、CPU で実行可能な CPU コード イメージが含まれています。このファイルは、ソフトウェアのコンパイラ/リンク ツールを使用して作成されます。ELF ファイルの作成方法については、ご使用のソフトウェア ツールのマニュアルを参照してください。

Data2MEM では、ELF ファイルが基本的なデータ入力形式として使用されます。ELF ファイルはバイナリ データを含むため、直接編集できません。Data2MEM には、既存の ELF ファイルの内容を検証する機能もあります。

詳細は、「コマンド ラインの使用」および「インプリメンテーション ツールの使用」を参照してください。

メモリ (MEM) ファイル

MEM ファイルは連続したデータ ブロックが記述されたテキスト ファイルで、MEM ファイルは直接編集できます。このファイルを編集する場合、Data2MEM にはコメントに対して // と /*...*/ の両方とも使用できます。

このファイルは、Data2MEM の入力および出力のどちらにもなります。

MEM ファイルのフォーマットは業界標準で、16 進数のアドレス指示子とデータ値の 2 つの基本的なエレメントが含まれます。アドレス指示子は @ で示され、その後に 16 進数のアドレス値が続きます。@ と最初のアドレス指示子の間はスペースなしです。

16 進数のアドレス指示子とその後に続く 16 進数のデータ値は、スペース、タブ、またはキャリッジ リターンで区切ります。データ値には、16 進数文字をいくつでも含めることができますが、16 進数文字の数が奇数である場合、最初に 0 が追加されます。たとえば、16 進数値は次のようになります。

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

これは、それぞれ次のように解釈されます。

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

16 進数値の前に 0x は使用できません。この接頭辞を MEM ファイルの 16 進数値の前に使用すると、構文エラーが発生します。

アドレスの後には、データ値を少なくとも 1 つは記述してください。データ値はその前に記述したアドレス値に属していればいくつでも記述できます。次は、一般的な MEM ファイル フォーマットの例です。

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

Data2MEM では、重複した記述するのを避けるため、連続したデータ ブロックを指定する場合は、アドレス指示子を最初に 1 度だけ指定するようにします。前述の例は、次のように記述できます。

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

アドレスが連続していれば、1 つ目以降のアドレス指示子は省略し、データ値はスペースで区切ります。ただし、これらの省略されたアドレスは、このファイルが入力ファイルとして使用されるか、出力ファイルとして使用されるかで異なります。入力メモリ ファイルと出力メモリ ファイルの違いについては、「出力ファイルとしてのメモリ ファイル」および「入力ファイルとしてのメモリ ファイル」を参照してください。

MEM ファイルには、連続したデータ ブロックをいくつでも含めることができます。データ ブロック間でアドレス範囲のサイズに違いがあってもかまいませんが、アドレス範囲は重複しないようにしてください。

Data2MEM では、次のメモリ タイプ キーワードが使用されます。

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

デバイスおよびプリミティブ別の有効なメモリ タイプについては、「デバイス ファミリ別ブロック RAM コンフィギュレーション」を参照してください。

出力としてのメモリ (MEM) ファイル

出力メモリ (MEM) ファイルは、主にサードパーティのメモリ モデルの Verilog シミュレーションで使用します。このファイル形式は、関連する業界の標準規格に準拠しています。

- すべてのデータ値のビット幅を同じにする必要があり、メモリ モデルで使用される幅と同じにする必要があります。
- データ値は、0 から開始する、より大きな配列内に含まれます。アドレス指示子は実際のアドレスではなく、データが開始される大きい方の配列の開始地点からのインデックス オフセットを示します。たとえば、次の MEM ファイルの一部分は、データが 16 ビット データ値の配列内にある 655 番目 (0 から開始) の 16 進数位置からデータが開始することを示しています。

  @654 24B7 6DF2 D897 1FE3 922A 5CAE 67F4...
- 2 つの連続したデータ ブロックの間にアドレスのギャップがある場合、このギャップ間のデータは定義されていないだけで、論理的には存在します。出力 MEM ファイルを生成する OUTPUT キーワードの使用については、「BMM ファイルの構文」を参照してください。

入力としてのメモリ (MEM) ファイル

入力 MEM ファイルには、次のような業界標準に従わないフォーマット制限があります。

- 隣接するデータ値間のスペースは無視されます。その代わり、連続するデータ ブロックの値はすべて、連続するビットストリームとして扱われます。Data2MEM では、ターゲットのブロック RAM がコンフィギュレーションされる幅に合わせて、このビットストリームをデータ値に分割します。隣接するデータ値間のスペースは、読みやすくする目的でのみ使用されます。
- アドレス指定子は、BMM ファイルで定義されたアドレス空間内にある必要があります。

  メモ : アドレス指定子は、CPU メモリ アドレスというよりも、BMM ファイルのアドレス空間に一致する値になります。
- アドレス指示子が実際には CPU メモリ アドレスではないという事実にも関わらず、連続するデータ値のアドレスは、値のバイト長によって異なります。その次に続くアドレスは、8 ビット値では 1、16 ビット値では 2、32 ビット値で 4 増加します。
- 2 つの連続したデータ ブロックの間にアドレスのギャップがある場合、このギャップは存在しないメモリとして扱われます。
- 2 つの隣接するデータ ブロック間でアドレス範囲が重複することはありません。
- 隣接するデータ ブロックは、BMM ファイルで定義された 1 つのアドレス空間内にある必要があります。

パリティを使用したメモリ (MEM) ファイル

パリティが使用されると、Data2MEM は上位ビット (MSB) のビット レーンがブロック RAM のパリティ データ ビットに接続されていると認識します。16 進数フォーマットでは値が偶数の 4 ビットのニブル値でしか定義できないので、パリティ ビットを追加するために、16 進数の桁を値の最上位に追加する必要があります。Data2MEM ではブロック RAM のデータ バス幅が認識されます。また、16 進数の 4 ビット幅は固定されているので、Data2MEM では 16 進数の 4 ビット幅の制限により、追加されたビットは削除されます。

たとえば、18 ビットのデータ値 0x23A24 は 16 進数では 20 ビットの値としてしか指定できません。この例の場合、最上位ニブルの最下位 2 ビット (ビット 17 と 16) に値 0x2 が含まれますが、そのニブルの最上位の 2 ビット (ビット 19 と 18) は使用されません。Data2MEM ではブロック RAM のデータ バスのデータ幅が 18 ビットであると認識されているので、これらの 2 つのデータ ビットが削除されます。同様に、9 ビットのデータ値 0x1D4 の場合は、最上位の 3 データ ビットが削除されます。こういった削除は、1 ビットや 2 ビットといった 4 ビット未満のビット幅のパリティ ビットを使用しないブロック RAM のデータ幅でも実行されます。

ビットストリーム (BIT) ファイル

BIT ファイル (ビットストリーム) は、バイナリ データ ファイルで、FPGA デバイスにダウンロードするビット イメージが含まれます。Data2MEM では、ザイリンクスのインプリメンテーション ツールを使用しなくても、BIT ファイルのブロック RAM データを置き換えることができるので、BIT ファイルは入力ファイルとしても出力ファイルとしても使用されますが、Data2MEM で変更できるのは、既存の BIT ファイルだけです。BIT ファイルは、最初にザイリンクスのインプリメンテーション ツールで作成されます。

BIT ファイルはバイナリ データであるため、直接編集できません。

Data2MEM を使用すると、BIT ファイルの内容を確認できます。詳細は、「コマンド ラインの使用」を参照してください。

メモ : Data2MEM は、暗号化または圧縮オプションを使用せずに作成されたビット ファイルをアップデートするためだけに使用できます。

Verilog ファイル

Verilog ファイル

- Data2MEM で出力されるテキスト ファイルで、拡張子は .v です。
- ブロック RAM を初期化する defparam レコードが含まれます。
- 主に合成前と合成後のシミュレーションに使用します。

Verilog ファイルは直接編集できますが、生成済みのファイルなので、直接編集することはお勧めしません。

VHDL ファイル

VHDL ファイル

- Data2MEM で出力されるテキスト ファイルで、拡張子は .vhd です。
- ブロック RAM を初期化する bit_vector 定数が含まれます。この定数は、インスタンシエート済みブロック RAM を初期化するジェネリック マップで使用されます。
- 主に合成前と合成後のシミュレーションに使用します。

VHDL ファイルは直接編集できますが、生成済みのファイルなので、直接編集することはお勧めしません。

UCF ファイル

UCF ファイル

- Data2MEM で出力されるテキスト ファイルです。
- ブロック RAM を初期化する INIT 制約が含まれます。

UCF ファイルは直接編集できますが、生成済みのファイルなので、直接編集することはお勧めしません。

XILINX®

第 3 章

BMM ファイルの構文

この章は Block RAM Memory Map (BMM) file で使用される構文について、次のセクションに分けて説明します。

- ブロック RAM メモリ マップ (BMM) の機能
- アドレス マップの定義 (複数プロセッサのサポート)
- アドレス スペースの定義
- バス ブロックの定義 (バス アクセス)
- ビット レーンの定義 (メモリ デバイスの使用)
- ブロック RAM メモリ マップ (BMM) ファイルの自動生成

ブロック RAM メモリ マップ (BMM) の機能

Block RAM Memory Map (BMM) file は、読みやすさを目的に作成されていますが、次の点でハイレベルなコンピュータ プログラミング言語と類似しています。

- キーワードまたはコマンドによるブロック構造。

  BMM は同じような構造をグループまたはデータ ブロック別に維持します。BMM は、アドレス スペース、バス アクセス グループ、コメントを記述するブロックを作成します。

- 記号名を使用。

  BMM はグループやエンティティを参照するための名前およびキーワードを使用し (読みやすさの改善)、アドレス スペース グループおよびブロック RAM を参照するための名前を使用します。

- コメントの記述。

  コメント ブロックは BMM ファイル内のどこにでも使用できます。

- 暗示的アルゴリズム。

  BMM ではデータ転送を半画像の形式で指定できるので、アドレスとブロック RAM の詳細なアルゴリズムを記述する必要性が軽減されています。これでソフトウェアがこのアルゴリズムを推論し、マップが実行できるようになります。

BMM では、次の表記規則を使用しています。

- キーワードの大文字と小文字の区別あり
- キーワードは大文字
- このインデントは読みやすくするためだけに挿入。推奨されるスタイルについては、「ブロック RAM メモリ マップ (BMM) ファイル」の BMM ファイル例を参照してください。
- スペースは、アイテムやキーワードを記述する場合以外は無視されます。
- 行末マークは無視されるので、1 行にいくつでもアイテムを含めることができます。
- コメントは、次のいずれかの方法で記述できます。
  - /*…*/

    文字、単語、行のコメント ブロックを囲むことができます。このタイプのコメントはネストできます。
  - //

    その行の終わりまでのすべてをコメントとして扱います。
- 数値は 10 進数または 16 進数として入力できます。16 進数では 0xXXX 形式を使用します。

BMM で使用されるバッカス ナウア構文への変更点についての詳細は、「BMM で変更されるバッカス ナウア記法構文」を参照してください。

アドレス マップの定義 (複数プロセッサのサポート)

Data2MEM では、次のキーワードを使用すると複数のプロセッサがサポートされます。

- **ADDRESS_MAP**
- **END_ADDRESS_SPACE**

このキーワードの構文は、次のとおりです。

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

これらのキーワードは、1 つのプロセッサのメモリ マップに含まれる ADDRESS_SPACE 定義を囲むように記述します。

- map_name は ADDRESS_MAP の中の ADDRESS_SPACE キーワードすべてを参照する識別子です。
- processor_type は、設計するプロセッサ タイプを指定します。
- iMPACT では、processor_ID が外部メモリの内容を適切なプロセッサにダウンロードするための JTAG ID として使用されます。

ADDRESS_MAP の定義はプロセッサごとに異なります。

ADDRESS_SPACE 名は、1 つの ADDRESS_MAP 内でほかと重ならないようにしてください。通常、インスタンス名は、1 つの ADDRESS_MAP 内でほかと重ならないようにする必要がありますが、Data2MEM では BMM ファイル全体の中で重ならないようにする必要があります。

アドレス タグで新しい形式が 2 つ採用され、以前に ADDRESS_SPACE 名が使用されていた箇所が変更されています。

- ADDRESS_SPACE は map_name と space_name 名で表され、cpu1.memory のようにピリオドで分割されます。
- アドレス タグ名は ADDRESS_MAP 名にだけ短縮され、データ変換はその ADDRESS_MAP 内の ADDRESS_SPACE にのみ限定されます。これは、各 ADDRESS_SPACE に名前を付けないで、特定のプロセッサにデータを送信する際に使用されます。

以前のバージョンとの互換性を持たせるため、ADDRESS_SPACE は前と同じように ADDRESS_MAP 構造の外で定義することもできます。これらの ADDRESS_SPACE キーワードはタイプ MB、PPC405 または PPC440 のプロセッサ「ID0」という名前の付いていない ADDRESS_MAP 定義に含まれていると判断されます。これらの ADDRESS_SPACE のアドレス タグは space_name として使用されます。

ADDRESS_MAP タグ名が付けられていない場合、データは一致するアドレス範囲を持つすべての ADDRESS_MAP の ADDRESS_SPACE ごとに変換されます。

アドレス スペースの定義

アドレス スペースの一番外側の部分は、次のように定義されます。

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

ADDRESS_SPACE と END_ADDRESS_SPACE ブロックのキーワードでは、1 つの連続するアドレス スペースが定義されます。ADDRESS_SPACE キーワードの後には、アドレス スペース全体を表す名前が必要です。「Data2MEM の使用方法」の「ソフトウェアおよびハードウェアのツール フロー」に示すように、アドレス スペース名の参照は、そのアドレス スペース全体の内容を参照することと同じです。

BMM ファイルには、同じアドレス スペースであっても、ADDRESS_SPACE の名前がほかと重なっていない限り、複数の ADDRESS_SPACE 定義を含めることができます。

アドレス スペース名の後には、どのタイプのメモリ デバイスから ADDRESS_SPACE が構築されるかを定義するキーワードが記述されます。これは、次のメモリ デバイス タイプのいずれかになります。

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

Spartan®-3 および Virtex®-4 デバイスの場合 :

- RAMB16 キーワードは、メモリをパリティを含まない 16Kbit のブロック RAM として定義します。
- RAMB18 キーワードは、メモリ スペースをパリティを使用した 18Kbit のブロック RAM として定義します。

Virtex-5 の場合 :

- RAMB32 キーワードは、ブロック RAM のメモリ サイズとスタイルがパリティを使用しないメモリとして定義します。
- RAMB36 キーワードは、パリティ メモリを使用した 36Kbit のブロック RAM を定義します。

キーワードは選択したメモリ サイズとスタイルに合わせて使用する必要があります。

MEMORY キーワードでは、メモリ デバイスが一般的なメモリとして定義されます。この場合、メモリ デバイスのサイズは ADDRESS_SPACE で定義されたアドレス範囲から決定されます。

COMBINED キーワードの詳細は、「BMM の自動生成」を参照してください。

メモリ デバイス タイプの後は、WORD_ADDRESSING キーワードを指定します (オプション)。WORD_ADDRESSING は Data2MEM に最小のアドレス指定可能なユニットがビット レーン幅であることを伝えます。WORD_ADDRESSING は、ブロック RAM でパリティを使用する場合に試用する必要があります。

この後には、アドレス ブロックが占めるアドレス範囲を [start_addr:end_addr] ペアで指定します。

start_addr の後に end_addr が記述されていますが、実際の順序はどちらでも構いません。どちらが先でも、Data2MEM は 2 つの値の小さい方を start_addr、大きい方を end_addr と認識します。

バス ブロックの定義 (バス アクセス)

ADDRESS_SPACE 定義の中には、「バス ブロック」というさまざまな数のサブブロック定義が含まれます。

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

各バス ブロックは、パラレルの CPU バス アクセスからアクセスされるブロック RAM ビット レーンの定義を含みます。「ブロック RAM のアドレス スペースのレイアウト例」には、それぞれ 8 ビットの 8 ビット ラインを含むバス ブロックが 4 行あります。

バス ブロックの指定される順序によって、バス ブロックがどのアドレス空間を使用するかが指定されます。最下位アドレスのバス ブロックが最初に定義され、最上位アドレスのバス ブロックが最後に定義されます。「ブロック RAM メモリ マップ (BMM) ファイル」の「BMM ファイルの例」では、最初のバス ブロックが CPU アドレスの 0xFFFFC000 ～ 0xFFFFCFFF を占めています。これは、「ブロック RAM がインプリメントされたアドレス スペースについての注意事項」の「ブロック RAM のアドレス スペースのレイアウト例」のブロック RAM の最初の行と同じです。2 つ目のバス ブロックは、CPU アドレスの 0xFFFFD000 ～ 0xFFFFDFFF を占め、図のブロック RAM の 2 行目に該当します。このパターンが最後のバス ブロックまで続きます。

バス ブロックが上から下に向かって定義されることで、Data2MEM がそれらのバスブロックをデータで満たす順序も制御されます。

ビット レーンの定義 (メモリ デバイスの使用)

ビット レーンを定義すると、CPU バス アクセスのどのビットがどのブロック RAM に割り当てられているかを指定できます。定義はそれぞれブロック RAM インスタンス名の形式で記述され、その後にビット レーンが使用するビット数が続きます。このインスタンス名の前にはシステム デザインで使用されるように、階層パスを付ける必要があります。構文は次のようになります。

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

パリティが使用されると、Data2MEM は上位ビット (MSB) のビット レーンがブロック RAM のパリティ データ ビットに接続されていると認識します。たとえば、[17:0] と定義されるビット レーンの場合、データ ビット 15:0 はブロック RAM の通常のデータ ビットに、ビット 16 と 17 はブロック RAM のパリティ ビットに接続されます。

通常、ビット数は次の順序で提供されます。

```
[MSB_bit_num:LSB_bit_num]
```

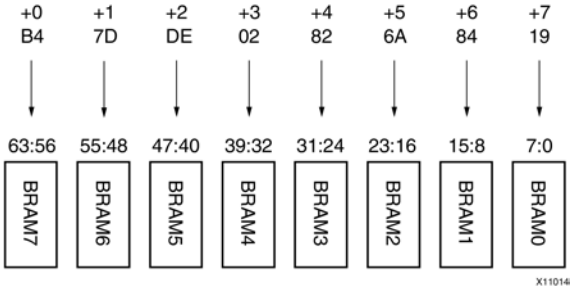
順序が LSB が最初で MSB が 2 つ目といったように逆になった場合、Data2MEM ではブロック RAM に [MSB_bit_num:LSB_bit_num] が入力される前にビット レーン値を逆ににします。

バス ブロックと同様、ビット レーンの定義される順序は重要です。ただし、ビット レーンの場合、この順序はビット レーンがバス ブロック CPU アクセスのどの部分を使用するかを示します。定義された最初のビット レーンは最上位のビット レーン値に、最後のビット レーンは最下位のビット レーン値になります。次の図の場合、最上位のビット レーンが BRAM7 に、最下位のビット レーンが BRAM0 になります。「ブロック RAM のアドレス スペースのレイアウト例」に示すとおり、これはビット レーンの定義される順序に対応しています。

Data2MEM がデータを入力すると、ビット レーン サイズの固まりでデータ入力ファイルから、一番右の値から一番左の値の順に取り出されます。たとえば、入力データの最初の 64 ビットが 0xB47DDE02826A8419 の場合、値 0xB4 がブロック RAM に入力される最初の値になります。

このビット レーンの順序では BRAM7 は 0xB4、BRAM6 は 0x7D になり、BRAM0 が 0x19 に設定されるまで、このように設定されます。。このプロセスは、メモリ スペースが満たされるか、または入力データがすべて挿入されるまで、バス ブロックが BRAM セットにアクセスするたびに繰り返されます。次は、最初のバス ブロックを展開して、このプロセスを示した図です。

ビット レーンにデータが入る順序



ビット レーンの定義は、ハードウェア コンフィギュレーションと一致している必要があります。BMM がハードウェアの実際の動作と異なって定義されていると、メモリ コンポーネントから取り出されたデータが不正になってしまいます。

ビット レーンの定義には、アドレス ブロックの定義で使用されるデバイス タイプ キーワードによって、オプションで構文が含まれることもあります。

RAMB16 ブロック RAM デバイスを指定する場合は、FPGA 内の物理的な行と列の箇所を指定できます。次は、物理的な行と列の箇所を指定した例です。

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

または

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

LOC キーワードは対応するブロック RAM を FPGA デバイスの特定位置に指定するために使用します。この例の場合、ブロック RAM は FPGA デバイスの 3 行目、5 列目に配置されます。PLACED キーワードは、バックアノテーションされた BMM ファイルを作成する際にザイリンクス インプリメンテーション ツールで挿入されます バックアノテーションされた BMM ファイルの詳細については、「ISE® インプリメンテーション ツールの使用」を参照してください。これらの定義はバス ビット値と行末を示すセミコロンの後に挿入されます。

OUTPUT キーワードは、次の形式でメモリ デバイスの MEM ファイルを出力するために挿入されます。これは次の形式で挿入されます。

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

このキーワードは、ビット レーン メモリ デバイスのデータ内容を含むメモリ (MEM) ファイルを作成するためのものです。出力ファイル名の最後には MEM ファイルの拡張子を付ける必要があります。ファイル パスはフルでも一部だけでも可能です。出力される MEM ファイルはシミュレーションが実行されると、デバイス メモリ モデルへの入力として使用されます。「ブロック RAM メモリ マップ (BMM) ファイル」の「BMM ファイルの例」に示すとおり、MEM ファイルは 2 つ目のバス ブロックのすべてのブロック RAM に対して作成されす。

ビット レーンとバス ブロックの定義の構文を正しく使用することとは別に、次のような制限もあります。

- 本書の例ではわかりやすくするためにデータ幅にはバイト幅だけを使用していますが、ブロック RAM のコンフィギュレーションにあわせてどのデータ幅でも同じ理論が適用されます。
- ビット レーンの番号は間があいたり、重複したりしないようにします。また、アドレス ブロックのすべてのビット レーンは同じビット幅にする必要があります。
- ビット レーン幅はデバイス タイプ キーワードで指定したメモリ デバイスで有効です。
- バス ブロックのビット レーン ブロック RAM で使用されるバイト ストレージの量は、バス ブロックの開始アドレスと終了アドレスで推論されるアドレス範囲と同じにする必要があります。
- すべてのバス ブロックのバイト数は同じサイズにする必要があります。
- ブロック RAM インスタンス名は 1 度だけ指定できます。
- バス ブロックには、1 つ以上のビット レーン定義が必要です。
- アドレス ブロックには、1 つ以上のバス ブロック定義が必要です。

Data2MEM では、これらすべての条件がチェック ボックスされ、違反があった場合はエラー メッセージが表示されます。

アドレス スペースの統合

BMM のアドレス空間とは、メモリ コントローラのことです。メモリ コントローラのメモリ デバイス生成を記述する BMM アドレス スペースは、メモリ コントローラごとに定義されます。

次のコード例は、16 ビット データ バスでコンフィギュレーションされたブロック RAM 2 つを含む 32 ビット バスのメモリ コントローラのアドレス空間 (4K) を示しています。

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

このコード例は、アドレス スペースとメモリ コントローラが 1:1 の関係にある限り有効ですが、現在のデザインでは、アドレス スペースとメモリ コントローラの関係を必ずしも 1:1 に維持する必要はありません。メモリ コントローラでは 2 のべき乗でバス アドレスをデコードするだけなので、2 のべき乗以外のメモリ サイズが必要な場合は、連続するアドレスを持つ複数のメモリ コントローラを使用する必要があります。

BMM ファイルには、16K と 32K のメモリ コントローラに対してアドレス スペースが 2 つに分けて定義されます・ Data2MEM では、ユーザーがこれらのアドレス スペースを論理的に 1 つであると認識させようとしても、物理的に 2 つの別々のアドレス スペースとして処理します。

Data2MEM ではデータを物理的な 16K または 32K アドレス スペースよりも大きい論理的な 48K のアドレス スペースに変換しようとすると、エラーが発生します。これは、データが複数のアドレス スペースに広がることができないためです。

この問題を回避するには、BMM アドレス スペースの構文で Data2MEM に複数の物理的なアドレス範囲を 1 つの論理的なアドレス スペースにまとめるように指定します。アドレス スペースをまとめるには、アドレス スペース ヘッダのデバイス タイプ キーワードを COMBINED というキーワードに書き換えます。

次の BMM コード例では、2 つの 32 ビット バス メモリ コントローラに対して 12K のアドレス スペースを記述しています。

- 1 つのメモリ コントローラは、16 ビット データ バスでコンフィギュレーションされたブロック RAM 2 つを含みます。
- もう 1 つのメモリ コントローラは、8 ビット データ バスでコンフィギュレーションされたブロック RAM 4 つを含みます。

最初のコード例とこのコード例の違いは、このコード例の後で説明します。

```
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;
```

2 つのコード例の違いは、次のとおりです。

- メモリ タイプにキーワード、COMBINED が使用されています。
- アドレス スペースのアドレス値は、論理的なアドレス スペース全体を示します。Data2MEM では、このアドレス スペースが複数の異なる物理的アドレス範囲から作成されていると認識されます。
- ブロック構造のキーワード、ADDRESS_RANGE と END_ADDRESS_RANGE が使用されています。この 2 つのキーワードで、すべての BUS_BLOCK とビット レーンを含む最初の例の ADDRESS_SPACE 定義と同じように、メモリ生成コンポーネントを囲みます。

アドレス範囲ヘッダには、アドレス範囲が生成されるメモリ コンポーネントのタイプ (この場合 RAMB16) が含まれます。Data2MEM は、最初のアドレス範囲を超えるデータを変換する際に、次のアドレス範囲を自動的に使用し、変換を続行します。

各アドレス範囲でそのメモリ コンポーネントが定義されるので、各アドレス範囲に対してブロック RAM、外部メモリ、フラッシュなどの異なるメモリ タイプを使用できることです。論理的なアドレス スペースに物理的メモリ タイプを混合できるので、メモリ オプションの柔軟性が広がります。

ブロック RAM メモリ マップ (BMM) ファイルの自動生成

Data2MEM では、ブロック RAM を 1 回インスタンシエーションすると、自動的に BMM ファイルが作成されます。この自動 BMM 機能を使用すると、シミュレーションを実行中に MEM ファイルのメモリ内容をザイリンクス ソフトウェアを再実行せずに変更できます。ただし、シミュレーション リコンパイルは必要になります。また、Data2MEM は最終 BIT ファイルに新しいメモリの変更を挿入するために実行する必要のある唯一のツールでもあります。

Data2MEM は、インスタンシエート済みブロック RAM で使用される INIT_FILE ジェネリックまたはパラメータに基づいて自動的に BMM ファイルを作成します。Data2MEM では、READ_WIDTH_A のブロック RAM 値を読み込んで、データ幅が決定されます。

データ幅が 8 以上の場合、Data2MEM ではパリティ ビットが MEM ファイルに含まれると認識します。

次に、IINIT_FILE ジェネリックまたはパラメータを使用した VHDL および Verilog コード例を示します。

VHDL の INIT_FILE コード例 :

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

Verilog の INIT_FILE コード例 :

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

XILINX®

第 4 章

コマンド ラインの使用

この章では、コマンド ラインの機能について次のセクションに分けて説明します。

- BMM ファイルの構文チェック
- データ ファイルの変換
- タグまたはアドレス ブロック名のフィルタを使用したデータ ファイルの変換
- ビットストリーム (BIT) ファイルのブロック RAM 変更
- BIT および ELF ファイルの内容の表示
- BMM のアドレス ブロック外部の ELF および MEM ファイルの無視
- アドレス スペースのテキスト出力ファイル

BMM の構文チェック

-bm オプションを使用すると、BMM ファイルの構文がチェックできます。次のコマンドを実行します。

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

Data2MEM が my.bmm という BMM ファイルを解析し、エラーまたは警告がある場合はそれを表示します。エラーも警告もない場合、BMM ファイルが問題ないことを示しています。

Data2MEM では、BMM の構文しかチェックされません。BMM ファイルがロジック デザインと合っているかどうかはユーザー自身が確認する必要があります。

データ ファイルの変換

-bm オプションと一緒に -bd オプションや -o オプションを使用すると、ELF (Executable and Linkable Format) ファイルやメモリ (MEM) データ ファイルを別のフォーマットに変換できます。

データ ファイルは Verilog と VHDL のブロック RAM 初期化ファイル、またはユーザー制約ファイル (UCF) のブロック RAM 初期化レコードに変換されます。次のコマンドを実行し、3 つすべての形式に変換します。

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

このコマンドを実行すると、次のファイルが生成されます。

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

ここではデータ ファイルは 1 つしか使用していませんが、-bd にデータ ファイルを指定すると、必要なだけファイルを変換できます。この後、これらのファイルは直接デザイン ソース ファイルに変換して、シミュレーション環境で使用できます。

ELF ファイルの内容をさまざまな方法でダンプしても変換できます。この方法でダンプを使用すると、効率的に ELF ファイルを MEM ファイルに変換できます。次のコマンドを実行します。

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

この code.mem ファイルには、バイナリの ELF ファイルの内容がテキスト形式で含まれます。このファイルは、ソースが入手できない ELF ファイルに変更を加える際に便利です。

ELF または MEM データ ファイルは、デバイス初期化 MEM ファイルに変換できます。入力データ ファイルの線形データはビット レーンを占めるデバイスの初期化 MEM ファイルに変換されます。これは、ブロック RAM と外部メモリ デバイスの両方に適用されます。次のコマンドを実行します。

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

ビット レーンは次のように表示されます。

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

出力される ram0.mem という MEM ファイルは、top/ram_cnlr/ram0 デバイスのみの初期化データを含みます。この機能が外部メモリ デバイスを使用してシミュレーション環境で暫定的に使用されます。

出力ファイル名 output は必要ではありますが、無視されます。出力ファイル名はビット レーンの OUTPUT 指示子で制御されます。

タグまたはアドレス ブロック名のフィルタを使用したデータ ファイルの変換

タグまたはアドレス ブロック名のフィルタリングを使用すると、データ ファイルの変換をさらに制御できます。-bd オプションを使用してアドレス ブロック名セットをリストすると、データ変換はそのアドレス ブロック セットにのみ限られます。-bd オプションは、次のように使用することもできます。

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

この方法では、code.elf のデータが別のアドレス ブロックと一致していても、データ変換はアドレス ブロック mem1 と mem2 でのみ実行されます。これにより、異なるデータ内容を同じアドレス範囲のアドレス ブロックに挿入できるようになります。

また、データ変換をデザインの一部にだけ制限し、ほかの部分はそのままにすることもできます。

タグ名でフィルタすると、-i オプションが使用され、アドレス スペースの不一致エラーが検出されないようになります。

ビットストリーム (BIT) ファイルのブロック RAM 変更

Data2MEM には、ザイリンクス インプリメンテーション ツールを再実行せずに、新しいブロック RAM データをビットストリーム (BIT) ファイルに挿入できる機能があります。この機能を新しい ELF および BMM ファイルと共に使用すると、Data2MEM が BIT ファイル イメージでブロック RAM の初期化をアップデートし、新規 BIT ファイルに出力します。タグ フィルタリングも使用できます。

次のコマンドを実行すると、new.bit という新しい BIT ファイルにが作成されます。このファイルでは、該当するブロック RAM の内容が code.elf ファイルの内容に置き換わります。

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

BMM ファイルには、各ブロック RAM に対して LOC または PLACED 制約が必ず含まれている必要があります。これらの制約は手動で追加もできますが、BitGen からアノテーションされた BMM ファイルには、既に含まれていることがほとんどです。詳細は、「ISE® Design Suite インプリメンテーション ツールの使用」を参照してください。

この方法で作成された新しい BIT ファイルを使用すると、インプリメンテーション ツールを再実行するよりもスピードが 100 倍から 1000 倍改善されます。この方法は、はじめはデザインのロジック部分が変更されていない場合にデザインに新しい CPU ソフトウェア コードを含めるために作られました。この方法を使用すると、ザイリンクス インプリメンテーション ツールを使用してコードを追加する必要はありません。

BIT および ELF ファイルの内容の表示

Data2MEM には、BIT (Bitstream) および ELF (Executable and Linkable Format) ファイルの内容を確認したり、ダンプしたりする機能があります。ダンプ内容は、入力ファイルが 16 進数のテキスト形式で記述されたもので、コンソールに表示されます。

-d オプションには次の 2 つのパラメータのいずれかを使用することで、データ ファイルのどの情報を表示するかが変更できます。

- e : 各セクションの追加情報を表示します。
- r : 重複する ELF ヘッダ情報を含めます。

次のコマンドを実行して ELF ファイルを表示します。

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

ELF ファイルには、Data2MEM のデータ変換に使用されるデータよりも多くのデータ (シンボル、デバッグ情報など) が含まれます。データ変換に Data2MEM で使用されるのは「Program header record」というセクションのデータのみです。

次のコマンドを実行して BIT ダンプを表示します。

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

ビットストリーム コマンドがそれぞれデコードされ、表示されます。ビット フィールド フラグを含むこれらのコマンドには、それぞれビット フィールドが記述されます。ブロック RAM 以外のデータを含むコマンドは、シンプルな 16 進数のダンプとして表示されます。ブロック RAM データはビットストリーム内でエンコードされるので、Data2MEM ではブロック RAM データをデコードされた 16 進数ダンプとして表示します。

これらのダンプはデバッグ目的で使用されていましたが、バイナリの ELF および BIT ファイルを比較する際にも便利です。

BMM のアドレス ブロック外部の ELF および MEM ファイルの無視

-i オプションを使用すると、BMM (Block RAM Memory Map) ファイル内のアドレス ブロック外にある ELF (Executable and Linkable Format) またはメモリ (MEM) ファイルのデータが Data2MEM で無視されるようになります。これにより、BMM ファイルで認識されるよりも多くのデータを含むデータ ファイルを使用できます。たとえば、Data2MEM はマスタ デザイン コード ファイルを、ファイルの一部のみがブロック RAM メモリの ELF コードになるデータであっても、ELF データ ファイルのように使用できます。

アドレス スペースのテキスト出力ファイル

-u オプションを使用すると、Data2MEM はデータがアドレス スペースに変換されていなくても、すべてのアドレス スペースに対してテキスト出力ファイルを生成します。ファイル タイプによって、出力ファイルは空白になるか、すべて 0 の初期化情報を含みます。このオプションを使用しない場合は、データが変換されたアドレス スペースのみが出力されます。

XILINX®

第 5 章

インプリメンテーション ツールの使用

この章では、ザイリンクス インプリメンテーション ツール フローに Data2MEM がどのように組み込まれているかについて説明します。次のセクションが含まれています。

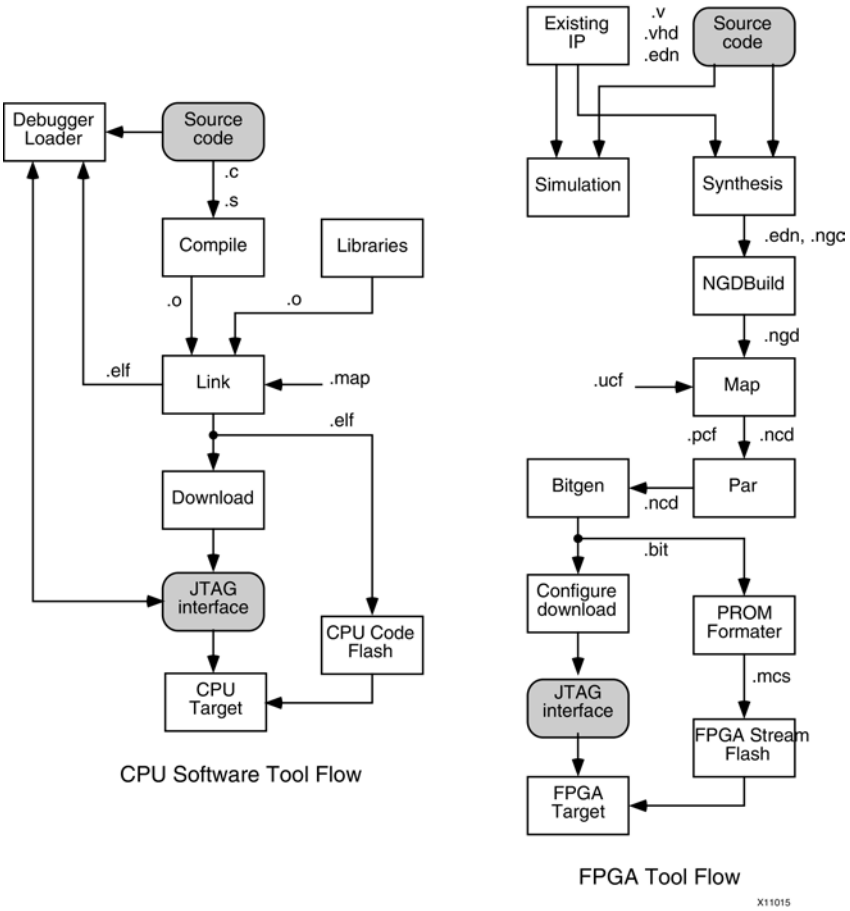
- NGDBuild の使用
- MAP および PAR の使用
- Bitgen の使用
- NetGen の使用
- FPGA Editor の使用
- iMPACT の使用
- インプリメンテーション ツールの制限

このフローを使用すると、ザイリンクス インプリメンテーション ツールから直接ブロック RAM のブロック RAM メモリ マップ (BMM) ファイルを関連付けることができます。

Data2MEM の機能を使用するには、NGDBuild、BitGen、NetGen、FPGA Editor の Data2MEM オプションのサブセットを使用します。

次の図は、ソフトウェア フローと使用されるファイルを示しています。

ソフトウェア フローと使用されるファイル



メモ : NGDBuild は、すべての入力デザイン ネットリストを変換し、その結果を 1 つのファイルに出力するプログラムです。NetGen は、シミュレーション用のネットリストが準備するコマンドです。

NGDBuild の使用

-bm オプションを使用すると、BMM ファイルの名前とパスを指定できます。

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

BMM ファイルを ISE® Design Suite プロジェクトに追加した場合、ISE Design Suite では BMM ファイルの変更が確認され、必要であればデザインがインプリメントし直されます。NGDBuild に入力される BMM ファイルには、LOC または PLACED キーワードは必要ありません。NGDBuild では、BMM ファイルの階層がソース ネットリストの階層と一致するかどうかがチェックされます。

NGDBuild は、Native Generic Database (NGD) ファイルの BMM_FILE プロパティを作成し、BMM ファイル デザインが使用されていることを伝えます。BMM ファイルが構文チェックされると、NGDBuild はその BMM ファイルに記述されたブロック RAM が実在しているかどうかを確認します (BMM ファイルの構文チェックには、コマンド ライン バージョンの Data2MEM でも実行できます)。また、BMM ファイルに記述されるブロック RAM の配置制約はすべて対応するブロック RAM に適用されます。

ISE Design Suite では、-bm オプションがサポートされます。NGDBuild でこのオプションが使用されるように設定する方法は、ISE Design Suite ヘルプの変換 (Translate) プロパティについての説明を参照してください。

MAP および PAR の使用

MAP および PAR (配置配線) へのコマンド ラインまたは機能の変更はありませんが、

ブロック RAM コンポーネントは、ユーザーが正しく接続する必要があります。間違って接続されたブロック RAM コンポーネントは MAP で削除されます。

ブロック RAM コンポーネントが削除されたかどうかは、MAP レポートの「Section 5 - Removed Logic」を確認してください。

BitGen の使用

-bd オプションでは、BMM ファイルで指定された ブロック RAM の生成に使用する ELF (Executable and Linkable Format) ファイルのパスとファイル名を指定します。

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

ELF ファイルに記述されたアドレス情報は、Data2MEM でどの ADDRESS_SPACE にデータを配置するか決定するために使用されます。

BitGen は、-bd オプションとそのファイル名、およびタグ情報すべてを Data2MEM に渡します。Data2MEM は NGDBuild で指定された BMM ファイルを処理し、ELF ファイルを使用して BMM 定義のブロック RAM ごとにブロック RAM 初期化文字列を作成します。この初期化文字列が Native Generic Database (NGD) ファイルをアップデートするために使用され、BIT (ビットストリーム) ファイルが作成されます。

各ブロック RAM の配置情報は、NCD ファイルで提供されます。BMM ファイルに記述されるブロック RAM の配置制約はすべて NCD に既に含まれています。その他すべてのブロック RAM には、前のツールの段階で配置制約が割り当てられています。これらの制約がバックアノテートされた BMM ファイルである <BMMfilename>_bd.bmm ファイルで Data2MEM に渡されます。このバックアノテートされた BMM ファイルには、PLACED キーワードでブロック RAM の配置位置が指定されています。この情報は、Data2MEM で必要とされます。

BitGen で読み込まれる NCD ファイルに BMM_fl プロパティは含まれていても -bd オプションが使用されていない場合でも、バックアノテーションされた BMM ファイルは生成されます。対応するブロック RAM の内容はすべて 0 になります。

このファイルには、元の BMM ファイルの情報に加えて、BMM ファイルで定義されたすべてのブロック RAM の配置情報も含まれます。この後、バックアノテーションされた BMM ファイルと出力される BIT ファイルを使用して、コマンド ライン バージョンの Data2MEM で BIT ファイルの置換を実行できます。

ISE® Design Suite では、-bd オプションがサポートされます。詳細は、
http://japan.xilinx.com/products/design_resources/design_tool/index.ditaを参照してください。

NetGen の使用

ISE® Design Suite では、-bd オプションと -bx オプションがサポートされます。NetGen 用にこれらのオプションを設定する方法については、http://japan.xilinx.com/products/design_resources/design_tool/index.dita を参照してください。

-bd オプションの使用

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

-bd オプションでは、BMM ファイルで指定された ブロック RAM の生成に使用する ELF ファイルのパスとファイル名を指定します。

ELF ファイルに記述されたアドレス情報は、Data2MEM でどの ADDRESS_SPACE にデータを配置するか決定するために使用されます。

1. NetGen は、-bd オプションとその <elf_filename> を Data2MEM に渡します。
2. Data2MEM は NGDBuild 中に指定された BMM ファイルを処理します。
3. ELF ファイルを使用して、制約付きブロック RAM それぞれのブロック RAM 初期化文字列を作成します。
4. この初期化文字列が NCD ファイルをアップデートするために使用され、BIT ファイルが作成されます。
5. NCD ファイルは、ブロック RAM 配置情報を Data2MEM に渡します。
6. この情報が <bramfilename>_bd.bmm ファイルを作成するのに使用されます。
7. このファイルには、すべてのブロック RAM の制約付きまたは制約なしの配置情報が含まれ、Data2MEM のコマンド ライン バージョンを使用できるようになります。

-bx オプションの使用

```
-bx [filepath]
```

-bx オプションでは、-bd オプションで指定された内容を元に、HDL シミュレーション用のメモリ デバイス MEM ファイルを出力するためのファイル パスを指定します。-bx オプションを使用する場合は、-bd オプションを必ず指定してください。

1. NetGen は、-bx オプションにその ファイル パス (オプション) を付けて Data2MEM に渡します。
2. Data2MEM は、MEM ファイルをそれぞれ指定されたファイル パスに出力します。ファイル パスが指定されていない場合は、デフォルトで現在作業中のディレクトリになります。ファイル パスが指定されている場合は、そのファイル パスが必ず存在している必要があります。ファイル パスは自動的には生成されません。
3. この結果出力されるネットリスト ファイルには、アノテーションされたブロック RAM インスタンスが含まれ、それぞれ INIT_FILE パラメータが指定されています。このパラメータでは、ブロック RAM を初期化するための MEM ファイルが指定されています。この後続くシミュレーションでは、INIT_FILE パラメータのメモリ ファイルが使用され、ブロック RAM の内容が初期化されます。
4. この機能は、現在のところ Virtex®-4 および Virtex-5 デバイスにのみ使用できます。

アップデートされた MEM ファイルは、-dx オプションを使用して Data2MEM をスタンドアロンで起動すると作成されます。これにより、アップデートされた MEM ファイルを生成するために NetGen を実行し直す必要がなくなります。

FPGA Editor の使用

-bd オプションでは、BMM ファイルで指定された ブロック RAM の生成に使用する ELF ファイルのパスとファイル名を指定します。

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

ELF ファイルに記述されたアドレス情報は、Data2MEM でどの ADDRESS_SPACE にデータを配置するか決定するために使用されます。

BMM で指定したブロック RAM は、FPGA Editor では読み込むことしかできません。FPGA Editor で BMM ファイルのブロック RAM の内容を変更をしても、書き出した NCD ファイルには反映されません。ブロック RAM の内容を変更するには、ELF ファイルを変更する必要があります。

ISE® Design Suite では、-bd オプションがサポートされます。詳細は、http://japan.xilinx.com/products/design_resources/design_tool/index.ditaを参照してください。

iMPACT の使用

Data2MEM の変換プロセスは、オンザフライで実行されます。iMPACT でダウンロードをすると、ビットストリーム (BIT) ファイルが FPGA デバイスにコンフィギュレーションされます。次は、その手順です。

1. iMPACT でコンフィギュレーションのダイアログ ボックスが開きます。
2. BMM を選択します。
3. Executable and Linkable Format (ELF) ファイルを選択します。
4. その ELF ファイルに関連させるタグ名を選択します。これにより、ELF ファイルの変換を選択したタグ名の ADDRESS_SPACE にのみ限定できます。
5. プロセッサごとにブート アドレスを入力します。

iMPACT プロセス フロー

iMPACT は、BIT ファイルを読み込んで、Data2MEM に次を渡します。

1. BIT ファイルのメモリ イメージ
2. BMM ファイル

3. ELF ファイル (タグ名付き)

- Data2MEM では、BMM ファイルの ADDRESS_SPACE キーワードに一致する ELF データを変換し、BIT ファイル メモリ イメージのブロック RAM の内容と置き換えます。このメモリ イメージが iMPACT に返されます。
- iMPACT ではアップデートされた BIT ファイルのメモリを FPGA デバイスにコンフィギュレーションし、プロセッサを停止します。
- iMPACT は Data2MEM からの外部メモリ データをリクエストします。Data2MEM は外部メモリ データをその開始アドレスとサイズ、該当するプロセッサの JTAG ID (複数プロセスのサポートについて説明したのと同様) と共に iMPACT に返します。iMPACT は、命令コマンドと共に JTAG を介してプロセッサにデータを送信します。
- 停止されたプロセッサは該当する外部メモリにデータを格納するためにバス サイクルを実行します。このプロセスは BMM ファイルで定義された ADDRESS_MAP 構造の外部メモリ データすべてが初期化されるまで繰り返されます。
- iMPACT は BMM ファイルで定義された ADDRESS_MAP 構造ごとに Data2MEM からブート アドレスをリクエストします。ブート アドレスは、最後の ELF ファイルから読み込まれ ADDRESS_MAP 構造に変換されるか、最初のコンフィギュレーション ダイアログ ボックスのオプションの一部として上書きされます。
- iMPACT はブート アドレスを設定し、各プロセッサを再び起動します。これでプロセッサがブート アドレスで開始されるようになります。

これは開発中に使用されるプロセスで、新しいテスト ソフトウェアをダウンロードする画期的な方法です。コンフィギュレーション ストリームを FPGA ではなくファイルに入れるように iMPACT に命令することで、同じプロセスを使用して SVF (Serial Vector Format) ファイルを生成することもできます。この後、iMPACT では SVF ファイルを ACE ファイルに変換できるようになります。このファイルが System ACE™ で使用されます。これにより、コンフィギュレーション ストリームが完全なコンフィギュレーション、ブロック RAM、外部メモリの初期化を含めた送信可能な形式になります。

プロセッサは必ず JTAG チェーンの最初のデバイスになり、JTAG ID が BMM ファイルの processor_ids と同じである必要があります。プロセッサの ADDRESS_MAP 構造が BMM ファイルに存在しない場合、そのプロセッサは初期化されません。

インプリメンテーション ツールの制限

次の制限は、Data2MEM 統合インプリメンテーション ツールを使用する際に適用されます。

- XDL はブロック RAM の初期化文字列をアップデートするために Data2MEM を呼び出すことはしないので、結果が FPGA Editor、BitGen、NetGen のものとは異なります。
- BMM ファイルで指定したブロック RAM は、間違って接続されていると MAP 中に削除されることがあります。この場合、Data2MEM が実行されたときにエラー メッセージが表示されます。
- CPU アドレスの物理的なブロック RAM アドレスへの変換は、HDL ハードウェア デザインの一部で実行しておく必要があります。

XILINX®

第 6 章

コマンド ライン構文およびオプション

この章では、コマンド ライン構文とそのオプションについて次のセクションに分けて説明します。

- コマンド ラインの構文
- コマンド ライン オプション
- BMM で変更されるバッカス ナウア記法構文

コマンド ラインの構文

```
<-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 ]>
```

コマンド ライン オプション

コマンド	説明

コマンド	説明
-bm *filename*	ブロック RAM メモリ マップ (BMM) 入力ファイルの名前を指定します。ファイルの拡張子を指定しない場合は、拡張子が .bmm のファイルが使用されます。このオプションを指定しない場合、ルート名が ELF (Executable and Linkable Format) または MEM (メモリ) ファイルと同じで拡張子が .bmm のファイルが使用されます。このオプションのみを指定すると、BMM ファイルの構文だけがチェックされ、エラーがレポートされます。-bm オプションは必要なだけ使用できます。
-bd *filename*	入力する ELF または MEM ファイルの名前を指定します。ファイルの拡張子を指定しない場合は、デフォルトで .elf になります。MEM ファイルの場合は、必ず拡張子 .mem を指定してください。 TagName が指定されている場合は、BMM ファイル内の同名のアドレス空間のみが変換に使用されます。TagName アドレス空間の外にあるその他入力ファイル データはすべて無視されます。ほかにオプションが指定されていない場合は、**-o u** *filename* が使用されます。-bd オプションは必要なだけ使用できます。 TagName には、次の 2 つのフォームがあります。 プロセッサ メモリ マップ (ADDRESS_MAP/END_add_MAP) の名前になります。これにより、1 つの名前でカプセル化された ADDRESS_SPACE のグループ全体を参照できます。プロセッサの TagName だけを指定すると、データ変換をそのプロセッサの ADDRESS_SPACE にのみ限定できます。名前の付いたプロセッサの TagName グループ内で特定の ADDRESS_SPACE を参照するには、そのプロセッサの TagName の後にピリオド、ADDRESS_SPACE の TagName を続けます。次に例を示します。 `cpu1.memory` 前のバージョンとの互換性を持たせるため、ADDRESS_MAP/END_add_MAP 構造外で定義される ADDRESS_SPACE はすべて暗示されるヌル プロセッサ名の中にカプセル化します。このため、これらの ADDRESS_SPACE は、TagName のようにその ADDRESS_SPACE 名だけで参照されます。 TagName には、次のキーワードがあります。 tag : データ ファイル名とアドレス スペース名を分けます。 boot : プロセッサのブート アドレスを含むデータ ファイルを識別します。tag キーワードよりも前に記述されます。boot キーワードの後にオプションの ADDRESS 値が使用される場合、その ADDRESS 値がデータ ファイルのブート アドレスを上書きします。各プロセッサの TagName グループごとに使用できる boot キーワードは 1 つだけです。プロセッサの TagName グループに対して boot キーワードが使用されない場合、最後の

コマンド	説明
	-bd オプションで指定したデータ ファイルがプロセッサのブート アドレスに使用されます。
-bx *filepath*	HDL シミュレーション用のメモリ デバイス MEM ファイルを出力するためのファイル パスを指定します。OUTPUT キーワードがビット レーンにある場合は、提供される MEM ファイル名が出力に使用されます。それ以外の場合、出力される MEM ファイルはアドレス スペースに数値が付いた名前になります。TagName が指定されている場合は、BMM ファイル内の同名のアドレス空間のみが変換に使用されます。TagName アドレス スペース外にあるその他入力ファイル データはすべて無視されます。-bx オプションは必要なだけ使用できます。 TagName には、次の 2 つのフォームがあります。 ・ プロセッサ メモリ マップ (ADDRESS_MAP/END_add_MAP) の名前になります。これにより、1 つの名前でカプセル化された ADDRESS_SPACE のグループ全体を参照できます。プロセッサの TagName だけを指定すると、データ変換をそのプロセッサの ADDRESS_SPACE にのみ限定できます。名前の付いたプロセッサの TagName グループ内で特定の ADDRESS_SPACE を参照するには、そのプロセッサの TagName の後にピリオド、ADDRESS_SPACE の TagName を続けます。次に例を示します。 `cpu1.memory` ・ 前のバージョンとの互換性を持たせるため、ADDRESS_MAP/END_add_MAP 構造外で定義される ADDRESS_SPACE はすべて暗示されるヌル プロセッサ名の中にカプセル化します。このため、これらの ADDRESS_SPACE は、TagName のようにその ADDRESS_SPACE 名だけで参照されます。 TagName には、次のキーワードがあります。 ・ tag : データ ファイル名とアドレス スペース名を分けます。 ・ boot : プロセッサのブート アドレスを含むデータ ファイルを識別します。tag キーワードよりも前に記述されます。boot キーワードの後にオプションの ADDRESS 値が使用される場合、その ADDRESS 値がデータ ファイルのブート アドレスを上書きします。各プロセッサの TagName グループごとに使用できる boot キーワードは 1 つだけです。プロセッサの TagName グループに対して boot キーワードが使用されない場合、最後の -bd オプションで指定したデータ ファイルがプロセッサのブート アドレスに使用されます。

コマンド	説明
-bt *filename*	入力するビットストリーム (BIT) ファイルの名前を指定します。ファイルの拡張子を指定しない場合は、拡張子が .bmm のファイルが使用されます。-o オプションを指定しない場合、出力 BIT ファイル名は、入力 BIT ファイルのルート名に _rp が付き、拡張子は .bit になります。これ以外の名前を付ける場合は、出力 BIT ファイル名を -o オプションで指定します。デバイス タイプは、BIT ファイルのヘッダから自動的に設定されるため、-p オプションの影響はありません。
-o*u/v/h/m/b/p/d filename*	出力ファイルの名前を指定します。`filename` の前の文字列は、出力されるファイルの形式を示します。このファイル タイプ文字の間にはスペースを入力できませんが、入力する順序は問いません。ファイル タイプ文字は、必要なだけ使用できます。ファイル タイプ文字は、それぞれ次を示しています。 · u = UCF ファイル形式 (拡張子は .ucf) · v = Verilog ファイル形式 (拡張子は .v) · h = VHDL ファイル形式 (拡張子は .vhd) · b = BIT ファイル形式 (拡張子は .bit) · p = 処理済の BMM 情報 (拡張子は .bmm) · d = ダンプ情報を示すテキスト ファイル (拡張子は .dmp) filename は、指定したすべての出力ファイル タイプに対して使用されます。ファイル拡張子が指定されない場合、最適なファイル拡張子が指定した出力ファイル タイプに追加されます。ファイル拡張子が指定されている場合、最適なファイル拡張子が残りのファイル形式に追加されます。出力ファイルには、変換されたすべての入力データ ファイルからのデータが含まれます。 **メモ**: メモ : 各メモリ デバイスに対して MEM ファイルは出力されなくなったので、ファイル タイプ文字の m は使用できません。MEM ファイルを出力するには、-bx オプションを使用してください。
-u	すべてのアドレス スペースに対して -o でテキスト出力をアップデートします。データがアドレス スペースに変換されていなくても、アップデートされます。ファイル タイプによって、出力ファイルは空白になるか、すべて 0 になります。このオプションを使用しない場合は、データが変換されたアドレス スペースのみが出力されます。

コマンド	説明
-mf *〈BMM info items〉*	BMM 定義を作成します。このオプションの後に記述するアイテムで、BMM ファイル内のアドレス スペース 1 つを定義します。アイテムはすべて、指定された順序で使用する必要があります。BMM ファイル内のアドレス スペースをすべて定義するためには、アイテム グループを必要な回数だけ使用します。-mf オプションは必要なだけ使用できます。 これらの定義は -bm ファイル オプションの代わりに使用するか、BMM ファイルを生成するために使用します。生成された BMM ファイルを出力するには、-o p filename オプションを使用します。この構文は 4 つのグループに含まれ、1 つの -mf オプションで組み合わせることができます。
-mf *〈MNAME MSIZE 〈MWIDTH [MWIDTH...]〉*	ユーザー メモリ デバイスの定義に使用される文字は、それぞれ次を示しています。 · m = 次の 3 つでユーザー メモリを定義できるので、BRAM 以外のメモリを必要に応じた大きさで、使用可能なコンフィギュレーション ビット幅で使用することができます。ユーザー メモリ デバイスの定義は、必ず使用前に同じ -mf オプションか、前述の別の -mf オプションのいずれかで定義します。 · MNAME = ユーザー定義メモリ デバイスの英数名を指定します。 · MSIZE = ユーザー メモリ デバイスの 16 進数サイズを指定します (例 : 0x1FFFF)。 · MWIDTH = ユーザー メモリ デバイスがコンフィギュレーションできるビット幅 (数値) を必要に応じて指定できます。値は 0 から開始します。
-mf〈p PNAME PTYPE PID 〈a ANAME ['x' | 'b'] ASTART BWIDTH 〈s MTYPE BSIZE DWIDTH IBASE〉	アドレス スペース定義に使用される文字は、それぞれ次を示しています。 · p = 次の 3 つのアイテムでアドレス マップを定義します。アドレス マップ定義は、必要なだけ繰り返します。アドレス マップ定義には、少なくともアドレス スペース定義が 1 つ必要です。 – PNAME = プロセッサ マップの英数名を指定します。 – PTYPE = アドレスマップのプロセッサ タイプを英数名で指定します。有効なプロセッサ タイプは、PPC405、PPC440、MB です。 – PID = アドレス マップの ID (数値) を指定します。 · a = 次の 3 つのアイテムで上記のアドレス マップ用のアドレス スペースを定義します。アドレス スペース定義は、必要なだけ繰り返します。アドレス スペース定義には、少なくともアドレス範囲定義が 1 つ必要です。 – ANAME = アドレス スペースの英数名を指定します。

コマンド	説明
	– ''x' | 'b' = アドレス スペース定義のアドレス指定方法を指定します。b の場合は、バイト アドレス指定方法になり、各 LSB がアドレス指定よりも 1 バイト前に増加するようになります。x の場合は、インデックス アドレス指定方法になり、各 LSB は、BWIDTH ビット サイズ値でアドレス指定よりも先に増加します。このアイテムはオプションなので、指定しない場合はバイト アドレス指定方法が使用されます。 – ASTART = アドレス スペースを開始する 16 進数アドレスを指定します (例 : 0xFFF00000)。 – BWIDTH = アドレス スペースのバス アクセスのビット幅を指定します。 – MTYPE : アドレス範囲を構成するメモリ タイプを指定します。使用できるメモリ タイプは、RAMB16、RAMB18、RAMB32、RAMB36 およびユーザー定義メモリ デバイスのいずれかです。 – BSIZE = アドレス範囲の 16 進数サイズを指定します (例 : 0x1FFFF)。 – DWIDTH = アドレス範囲内の各ビット レーンのビット幅を指定します。 – IBASE = 各ビット レーン デバイスに割り当てられた階層/パーツ インスタンスのベース名 (英数字)。インスタンス名をすべて異なるものにするには、右側に数値を足していきます。
-mf <tTNAME ['x'|'b']ASIZE BWIDTH<s MTYPE BSIZE DWIDTH IBASE>	アドレス テンプレート定義のアイテムは、それぞれ次を示しています。 · t = 次の 3 つのアイテムでアドレス テンプレートを定義します。これにより、アドレス スペースのテンプレートが定義できます。1 度テンプレートを作成しておくと、複数のアドレス スペースに使用できます。 · TNAME = アドレス テンプレートの英数名を指定します。 · 'x' | 'b' = アドレス テンプレート定義のアドレス指定方法を指定します。 · b の場合は、バイト アドレス指定方法になり、各 LSB がアドレス指定よりも 1 バイト前に増加するようになります。 · x の場合は、インデックス アドレス指定方法になり、各 LSB は、BWIDTH ビット サイズ値でアドレス指定よりも先に増加します。このアイテムはオプションなので、指定しない場合はバイト アドレス指定方法が使用されます。 · s = 次の 4 つのアイテムで上記のアドレス スペース用のアドレス範囲を定義します。アドレス範囲定義は、必要なだけ繰り返します。

コマンド	説明
	· SIZE = アドレス テンプレートの 16 進数サイズを指定します (例 : 0x1FFFF)。 · BWIDTH = アドレス テンプレートのバス アクセスのビット幅を指定します。 · MTYPE : アドレス範囲を構成するメモリ タイプを指定します。使用できるメモリ タイプは、RAMB16、RAMB18、RAMB32、RAMB36 およびユーザー定義メモリ デバイスのいずれかです。 · BSIZE = アドレス範囲の 16 進数サイズを指定します (例 : 0x1FFFF)。 · DWIDTH = アドレス範囲内の各ビット レーンのビット幅を指定します。 · IBASE = 各ビット レーン デバイスに割り当てられた階層 /パーツ インスタンスのベース名 (英数字)。インスタンス名をすべて異なるものにするには、ルート インスタンス名の右側に数値を足していきます。値は 0 から開始します。また、ベース インスタンス パスは */ 構文で開始できます。テンプレートがインスタンスに広がる場合、インスタンスの IROOT と IBASE がまとめられ、完全なインスタンス パスが作成されます。
-mfi TNAME INAME ASTART IROOT	アドレス インスタンス定義のアイテムは、それぞれ次を示しています。 · i = 次の 4 つのアイテムでアドレス インスタンスを定義します。これにより、アドレス スペースのテンプレート以外の変動部分が定義できます。このインスタンス アイテムをアドレス テンプレートに適用すると、新しいアドレス インスタンスを作成できます。 · ANAME = アドレス テンプレートの英数名を指定します。 · INAME = アドレス インスタンスの英数名を指定します。 · ASTART = アドレス スペースを開始する 16 進数アドレスを指定します (例 : 0xFFF00000)。 IROOT = 各ビット レーン デバイスに割り当てられた階層 /パーツ インスタンスのルート名 (英数字)。詳細は、IBASE の説明を参照してください。

コマンド	説明
-pp filename	入力するプリプロセス ファイルの名前を指定します。ファイルの拡張子を指定しない場合は、拡張子が .bmm のファイルが使用されます。-o p filename オプションが使用されると、プリプロセスされた出力が指定したファイルに送信されます。ファイルの拡張子を指定しない場合は、拡張子が .bmm のファイルが使用されます。-o p filename オプションが使用されない場合は、プリプロセスされた出力がコンソールに送信されます。入力ファイルは BMM ファイルである必要はなく、テキスト形式のファイルであればどれでも入力ファイルとして使用できます。
-p partname	ターゲットの Virtex®-4 または Virtex-5 パーツ名。このオプションを指定しない場合は、デフォルトで xcv50 パーツが使用されます。-h オプションを使用すると、サポートされるパーツ名のリストが表示されます。
-d e|r	入力 ELF または BIT ファイルの内容をフォーマット済みのテキスト レコードとしてダンプします。BIT ファイルのダンプは、BIT ファイル コマンドと各ブロック RAM を表します。ELF ファイルをダンプする場合は、-d オプションの後に 2 つの修飾文字を使用することもできます。この修飾文字の間にはスペースを入力できませんが、入力する順序は問いません。この修飾文字は指定された期間に何回でも使用できます。 これらの修飾文字は、次のとおりです。 · e = EXTENDED モード。各 ELF セクションの追加情報を表示します。 · r = RAW モード。重複する ELF 情報も含めます。
-i	BMM ファイルで定義されたアドレス スペース外にある ELF または MEM データを無視します。このオプションを指定しないと、エラーが発生します。
-f filename	オプション ファイルの名前を指定します。ファイルの拡張子を指定しない場合は、デフォルトで .opt になります。これらのオプションはコマンド ライン オプションと同じですが、テキスト ファイルに記述されているところが違います。オプションとそのアイテムは、同じテキスト ラインに表示されるはずです。また、同じテキスト ラインに必要なだけオプションを含めることができます。このオプションが使用できるのは 1 度のみで、OPT ファイルに -f オプションを含めることはできません。

コマンド	説明
-g e|w|i	Data2MEM のメッセージが出力されないようにします。このオプションの後の文字で、どのタイプのメッセージを表示されないようにするか指定できます。このメッセージ タイプの間にはスペースを入力できませんが、文字の入力する順序は問いません。このメッセージ タイプ文字は指定された期間に何回でも使用できます。メッセージ タイプ文字の使用はオプションです。メッセージ タイプを空白にすると、-q wi が使用されます。メッセージ タイプ文字は、それぞれ次を示しています。 · e = エラー メッセージを非表示 · w = 警告メッセージを非表示 · i = 情報メッセージを非表示
-h-h	ヘルプ テキストとサポートされるパーツ名のリストを表示します。

BMM で変更されるバッカス ナウア記法構文

```
Address_block_keyword ::= “ADDRESS_SPACE”;
End_add_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 ::= "RAMB16"|"RAMB18"|"RAMB32"|"RAMB36"|"MEMORY"|"COMBINED";

Number_range ::= “[“ NUM “:” NUM “]”;

Name_path ::= IDENT ( “/” IDENT )*;

BRAM_instance_name ::= Name_path;

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

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

Bit_lane_def ::= BRAM_instance_name Number_range
            [ BRAM_location_spec | MEM_output_spec ]“;” ;

Bus_block_def ::= Bus_block_keyword
            ( Bit_lane_def )+
            End_bus_block_keyword“;” ;

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