

エンベデッド システム ツール リファレンス マニュアル

UG1043 (v2019.2) 2019 年 10 月 30 日

この資料は表記のバージョンの英語版を翻訳したもので、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。

改訂履歴

2019/10/30: Vivado® Design Suite 2019.2 リリース。2018.2 リリースから内容の変更なし。

セクション	改訂内容
2018 年 6 月 6 日 バージョン 2018.2	
全体的なアップデート	編集上の更新のみ。技術内容の変更なし。
2018 年 4 月 4 日 バージョン 2018.1	
全体的なアップデート	Vivado® Design Suite 2018.1 リリース。2016.1 リリースから内容の変更なし。

目次

改訂履歴	2
第 1 章: エンベデッド システムとツールの概要	
設計プロセスの概要	5
Vivado Design Suite の概要	7
ソフトウェア開発キット (SDK)	8
第 2 章: GNU コンパイラ ツール	
概要	10
コンパイラのフレームワーク	11
コンパイラの使用法とオプション	12
MicroBlaze コンパイラの使用法とオプション	25
Arm Cortex-A9 コンパイラの使用法とオプション	40
その他の注意事項	42
第 3 章: Xilinx System Debugger	
SDK System Debugger	43
Xilinx System Debugger コマンドライン インターフェイス (XSDB)	44
第 4 章: フラッシュ メモリのプログラム	
概要	45
プログラム フラッシュ ユーティリティ	46
その他の注意事項	48
付録 A: GNU ユーティリティ	
MicroBlaze プロセッサ用の汎用ユーティリティ	53
MicroBlaze プロセッサ用ユーティリティ	53
その他のプログラムおよびファイル	55
付録 B: その他のリソースおよび法的通知	
ザイリンクス リソース	56
ソリューション センター	56
Xilinx Documentation Navigator およびデザイン ハブ	56
参考資料	57
トレーニング リソース	57
お読みください: 重要な法的通知	58

エンベデッド システムとツールの概要

この章では、MicroBlaze™ エンベデッド プロセッサおよび Cortex A9、A53、R5 ARM プロセッサをベースにしたシステムを開発するために、ザイリンクス Vivado® Design Suite に含まれているエンベデッド システム ツールと開発フローについて説明します。

Vivado Design Suite は、ザイリンクス FPGA デバイスにインプリメントする完全なエンベデッド プロセッサ システムを設計するためのツールです。

Vivado Design Suite は、デザインをザイリンクスのプログラマブル ロジック デバイスにインプリメントするために必要な開発システム ツールで、次のものが含まれています。

- Vivado IP ツール。エンベデッド プロセッサ ハードウェアの開発に使用します。
- SDK (ソフトウェア開発キット)。Eclipse オープン ソース フレームワークに基づくツールで、エンベデッド ソフトウェア アプリケーションの開発に使用できます。SDK はスタンドアロン プログラムとしても利用できます。
- プロセッサやペリフェラルなどのエンベデッド プロセッサ IP コア

Vivado 関連の資料およびその他の情報へのリンクは、[付録 B 「その他のリソースおよび法的通知」](#)を参照してください。

設計プロセスの概要

Vivado に含まれるツールを使用すると、図 1-1 に示すように、エンベデッド システムの設計フローを最初から最後まで実行できます。

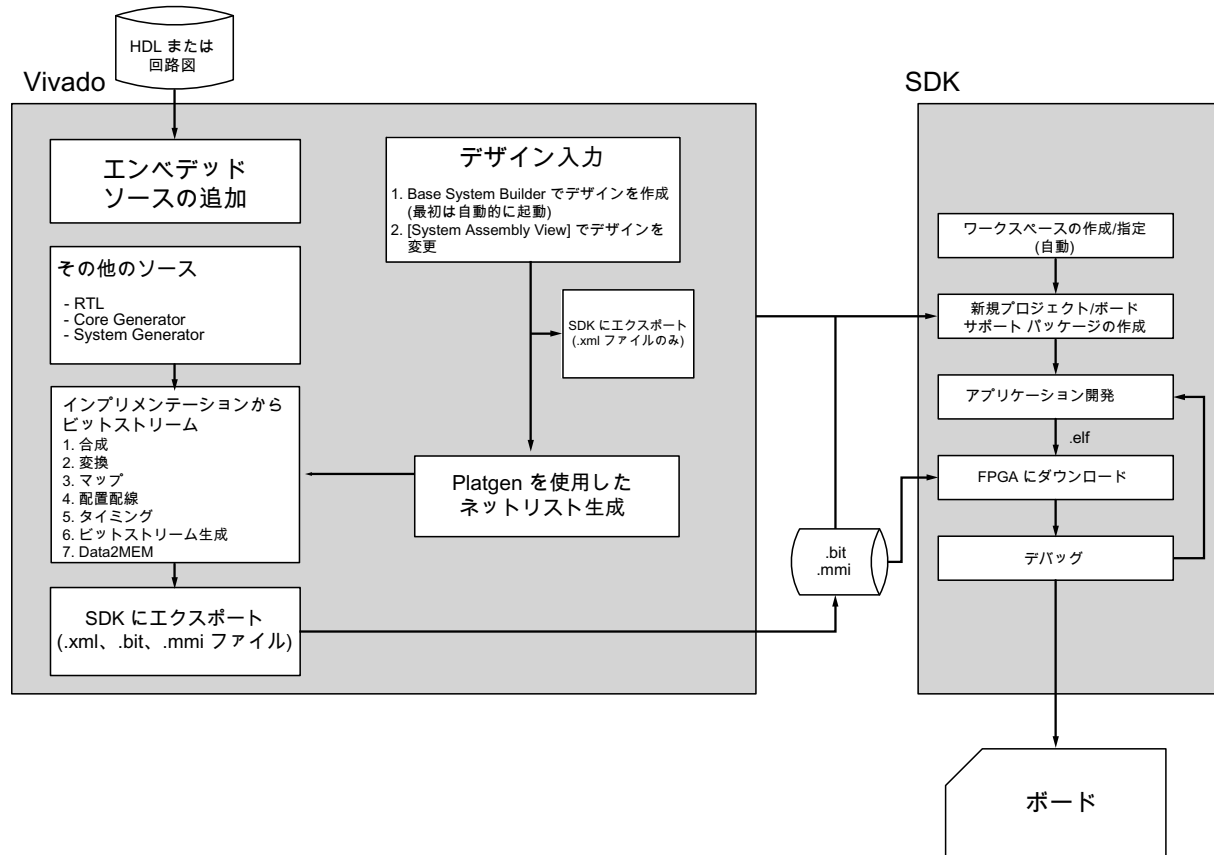


図 1-1: エンベデッド システム設計プロセスのフロー

ハードウェア開発

ザイリンクスの FPGA 技術を使用すると、プロセッサ サブシステムのハードウェア ロジックをカスタマイズできます。標準的な既製のマイクロプロセッサチップまたはコントローラチップでは、このようなカスタマイズは不可能です。

「ハードウェア プラットフォーム」という用語は、ザイリンクスの技術を使用して作成する柔軟なエンベデッド プロセッサ サブシステムを指します。

ハードウェア プラットフォームは、プロセッサ バスに接続された 1 つまたは複数のプロセッサおよびペリフェラルで構成されています。

ハードウェア プラットフォームの記述が完了したら、これをエクスポートして SDK で使用することができます。

ソフトウェア開発

ボード サポート パッケージ (BSP) はアプリケーションを構築するためのソフトウェア ドライバーおよび OS をまとめたものです (OS はオプション)。作成するソフトウェア イメージには、ザイリンクス ライブラリのうちエンベデッド デザインで使用するもののみが含まれます。BSP で実行するアプリケーションを複数作成できます。

ソフトウェア アプリケーションおよび BSP を作成する前に、ハードウェア プラットフォームを SDK にインポートしておく必要があります。

検証

Vivado には、ハードウェアとソフトウェア両方の検証ツールが含まれています。使用可能な検証ツールは、次のとおりです。

シミュレーションを使用したハードウェア検証

ハードウェア プラットフォームの機能を検証するには、シミュレーション モデルを作成して HDL シミュレータで実行します。システムをシミュレーションすると、プロセッサはソフトウェア プログラムを実行します。シミュレーション モデルのタイプには、ビヘイビア、構造、またはタイミングのいずれかを選択し、モデルを作成できます。

デバッグによるソフトウェア検証

ソフトウェア検証には、次の方法があります。

- サポートされている開発ボードにデザインを読み込み、デバッグ ツールでターゲット プロセッサを制御します。
- コード実行のプロファイルを作成して、システムのパフォーマンスを評価します。

デバイスのコンフィギュレーション

ハードウェア プラットフォームおよびソフトウェア プラットフォームが完成したら、FPGA デバイス用にコンフィギュレーション ビットストリームを作成します。

- プロトタイプを作成する場合は、ホスト コンピューターに接続した状態で、ビットストリームをエンベデッド プラットフォームで実行するソフトウェアと共にダウンロードします。
- プロダクション用の場合は、コンフィギュレーション ビットストリームとソフトウェアを FPGA に接続した不揮発性メモリに保存します。

Vivado Design Suite の概要

エンベデッド ハードウェア プラットフォームは通常、プロセッサ バスで接続された 1 つまたは複数のプロセッサ、ペリフェラル、およびメモリ ブロックで構成されています。また、デバイスの外部に接続するポートもあります。各プロセッサ コア (pcore またはプロセッサ IP と呼ばれる) には、その動作をカスタマイズするためのパラメーターがあります。これらのパラメーターはペリフェラルおよびメモリのアドレス マップも定義します。IP インテグレーターではさまざまなオプションの機能を選択できるので、FPGA には作成するアプリケーションに必要な機能のサブセットをインプリメントするだけで済みます。

図 1-2 は Vivado アーキテクチャ構造の概要を示したもので、エンベデッド システムを作成するため各ツールの関連性を表しています。

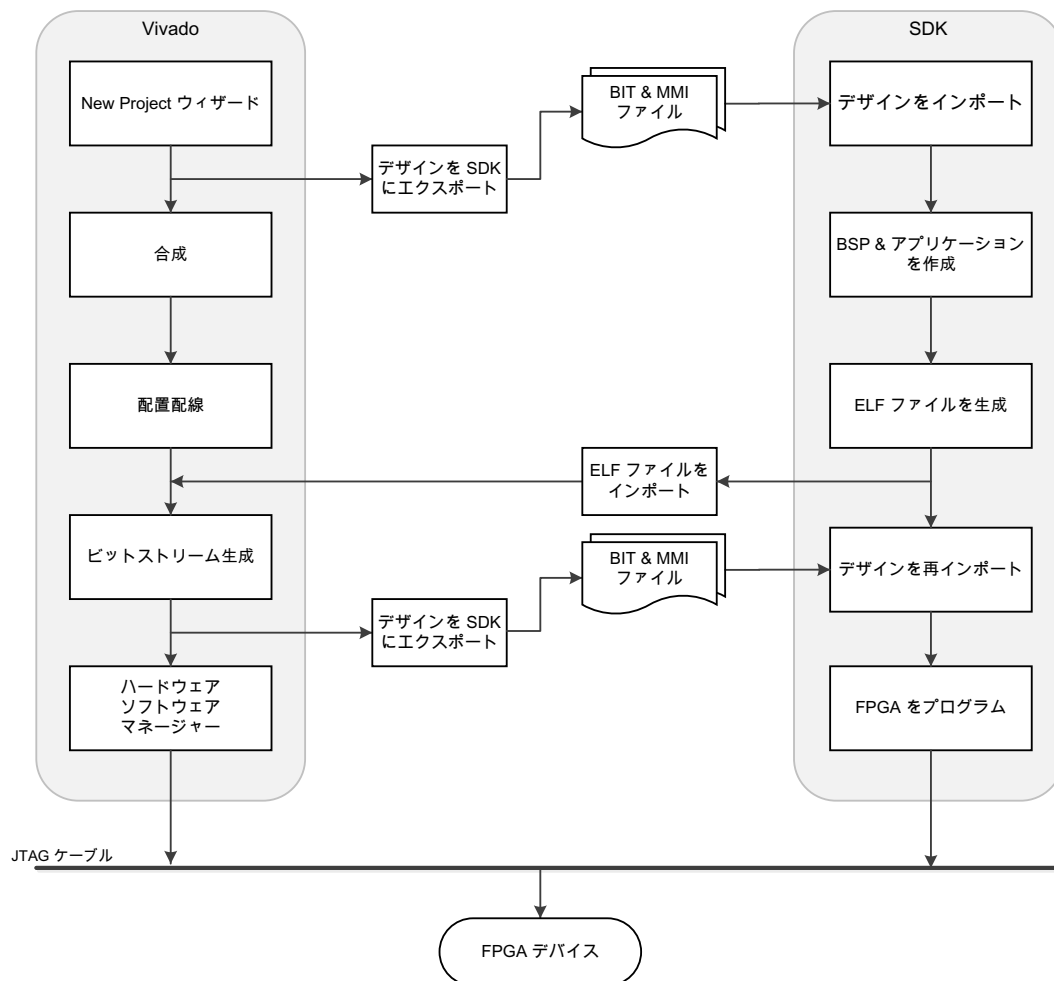


図 1-2: Vivado Design Suite ツールのアーキテクチャ

ソフトウェア開発キット (SDK)

SDK (ソフトウェア開発キット) は、ソフトウェア アプリケーション プロジェクトの開発環境で、Eclipse オープンソースをベースにしています。SDK には、次の機能があります。

- Vivado から独立した形でインストールされ、ディスク容量使用率が小さい
- シングルまたはマルチプロセッサ システムでのソフトウェア アプリケーション開発をサポート
- Vivado で生成されるハードウェア プラットフォーム定義をインポート
- チーム環境でのソフトウェア アプリケーション開発をサポート
- サードパーティ OS 用のボード サポート パッケージ (BSP) を作成およびコンフィギュレーション
- ハードウェアおよびソフトウェアの機能をテストするための標準サンプル ソフトウェア プロジェクトを提供
- ソフトウェア アプリケーション用リンカー スクリプトの生成、FPGA デバイスのプログラム、パラレル フラッシュ メモリのプログラムのための簡単な GUI インターフェイス
- 多機能な C/C++ コード エディターおよびコンパイル環境
- プロジェクト管理機能
- アプリケーションのビルドをコンフィギュレーションし、makefile の生成を自動化
- エラー ナビゲーション
- エンベデッド ターゲットのデバッグおよびプロファイル作成をスムーズに行う統合環境

SDK の詳細は、SDK ヘルプ [参照 1] を参照してください。

表 1-1: ソフトウェア開発および検証ツール

GNU コンパイラ ツール	作成されたプラットフォームに基づき、ソフトウェア アプリケーションを作成します。
Xilinx System Debugger (XSDB)	hw_server およびその他の TCF サーバー用のコマンドライン インターフェイス。
SDK System Debugger	シミュレーション モデルまたはターゲット デバイスでソフトウェアをデバッグするための GUI を提供します。
プログラムフラッシュユーティリティ	ソフトウェアおよびデータを使用してオンボードのシリアルおよびパラレルフラッシュ デバイスを削除してプログラムできるようにします。

GNU コンパイラ ツール

システムに含まれる各プロセッサのアプリケーション実行ファイルをコンパイルおよびリンクするには、GNU コンパイラ ツールが呼び出されます。プロセッサ別のコンパイラは次のとおりです。

- MicroBlaze プロセッサの場合は mb-gcc コンパイラ。
- ARM プロセッサの場合は arm-none-eabi-gcc、arm-linux-gnu-eabi-gcc、aarch64-linux-gnu-gcc、aarch64-none-eabi-gcc、armv5-none-eabi-gcc コンパイラ。

エンベデッド ツールのアーキテクチャ概要 (7 ページの図 1-2) に示すように、GNU コンパイラ ツールは次の処理を実行します。

- コンパイラ: ターゲット プロセッサ用の C ソース ファイルとヘッダー ファイルまたはアセンブラー ソース ファイルを読み込みます。
- リンカー: コンパイルされたアプリケーションと選択されたライブラリをまとめ、ELF フォーマットの実行ファイルを作成します。リンカーは、リンカー スクリプトも読み込みます。ツールで生成されたデフォルトのもの、またはユーザーが作成したものが読み込まれます。

GNU コンパイラ ツールおよびユーティリティの詳細は、第 2 章「GNU コンパイラ ツール」および付録 A「GNU ユーティリティ」を参照してください。

Xilinx System Debugger (XSDB)

Xilinx System Debugger (XSDB) は、hw_server およびその他の TCF サーバー用のコマンドライン インターフェイスです。XSDB は TCF サーバーと通信するので、TCF サーバーでサポートされる機能を最大限に活用できます。

XSDB では、FPGA のプログラム、ターゲットへのプログラムのダウンロードと実行、およびその他の高度な機能がサポートされます。詳細は、第 3 章「Xilinx System Debugger」を参照してください。

SDK System Debugger

SDK System Debugger は、オープン ソース ツールからザイリンクスによりカスタマイズされて作成されたツールで、ザイリンクス SDK に統合されています。この SDK デバッガーを使用すると、プログラムの実行中に何が起きているかを確認できます。プロセッサを停止するブレークポイントまたはウォッチポイントの設定、プログラムのステップ実行、プログラム変数およびスタックの確認、システム内のメモリ内容の確認などを実行できます。

SDK デバッガーでは、Xilinx System Debugger (XSDB) を使用したデバッグがサポートされます。詳細は、第 3 章「Xilinx System Debugger」を参照してください。

注記: GDB フローは廃止予定となっており、今後のデバイスでは使用できなくなる予定です。System Debugger は Digilent ケーブルを使用した ARM での使用のみに使用できます。

プログラム フラッシュ ユーティリティ

プログラム フラッシュ ユーティリティは、汎用性を持たせて、さまざまなフラッシュ デバイスおよびレイアウトに対応するように設計されています。詳細は、第 4 章「フラッシュ メモリのプログラム」を参照してください。

GNU コンパイラ ツール

概要

Vivado® Design Suite には、MicroBlaze™ プロセッサおよび Cortex A9 プロセッサ用の GNU コンパイラ コレクション (GCC) が含まれています。

- Vivado GNU ツールでは、C および C++ 言語の両方がサポートされています。
- MicroBlaze 用の GNU ツールには、mb-gcc および mb-g++ コンパイラ、mb-as アセンブラー、mb-ld リンカーが含まれています。
- Cortex A9 Arm プロセッサ用のツールには、arm-xilinx-eabi-gcc および arm-xilinx-eabi-g++ コンパイラ、arm-xilinx-eabi-as アセンブラー、arm-xilinx-eabi-ld リンカーが含まれています。
- また、C、数学、GCC、および C++ の標準ライブラリも含まれています。

コンパイラでは、アセンブラー、リンカー、オブジェクト ダンプなど、共通バイナリ ユーティリティ (binutils と呼ばれる) も使用されます。MicroBlaze および Arm 用のコンパイラ ツールでは、バージョン 2.16 の GNU に基づく GNU binutils が使用されます。概念、オプション、使用法、言語およびライブラリ サポートの例外については、[付録 A 「GNU ユーティリティ」](#) で説明します。

コンパイラのフレームワーク

このセクションでは、MicroBlaze プロセッサと Cortex-A9 Arm プロセッサの両方のコンパイラに共通した機能について説明します。図 2-1 に、GNU ツールフローを示します。

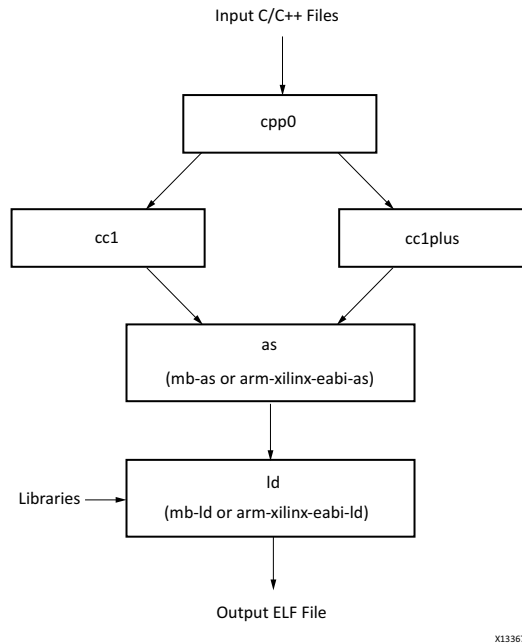


図 2-1: GNU ツール フロー

GNU コンパイラの名前は、MicroBlaze 用は mb-gcc、Arm コア用は arm-xilinx-eabi-gcc です。GNU コンパイラはラッパーで、次のプログラムを呼び出します。

- プリプロセッサ (cpp0)
コンパイラにより最初に呼び出され、すべてのマクロをソース ファイルおよびヘッダー ファイルでの定義に置き換えます。
- マシンおよび言語別のコンパイラ
プリプロセスされたコード、つまり第 1 段階で出力されたコードに対して実行します。言語別のコンパイラには次の 2 つがあります。
 - C コンパイラ (cc1)
入力 C コードの最適化の大部分を実行し、アセンブリ コードを生成します。
 - C++ コンパイラ (cc1plus)
入力 C++ コードの最適化の大部分を実行し、アセンブリ コードを生成します。
- アセンブラ (MicroBlaze 用の mb-as、Arm 用の arm-xilinx-eabi-as)
アセンブラー コードには、アセンブリ言語のニーモニックが含まれます。アセンブラーは、これらのニーモニックを機械語に変換します。コンパイラで生成されたラベルの一部も解決し、オブジェクト ファイルを生成してリンカーに渡します。

- リンカー (MicroBlaze 用の mb-ld、Arm 用の arm-xilinx-eabi-ld)
アセンブラーで生成されたオブジェクト ファイルをリンクします。コマンド ラインでライブラリが指定されている場合は、アセンブラーからの関数をリンクすることにより、コード内の未定義の参照の一部を解決します。

実行オプションについては、次のセクションで説明します。

- [15 ページの「よく使用されるコンパイラ オプション: 早見表」](#)
- [19 ページの「リンカー オプション」](#)
- [25 ページの「MicroBlaze コンパイラ オプション: 早見表」](#)
- [32 ページの「MicroBlaze リンカー オプション」](#)
- [40 ページの「Arm Cortex-A9 コンパイラの使用法とオプション」](#)

注記: この章では、「GCC」と表記されている場合は MicroBlaze のコンパイラである mb-gcc、「G++」と表記されている場合は MicroBlaze の C++ コンパイラである mb-g++ を指します。

コンパイラの使用法とオプション

使用法

GNU コンパイラを実行するには、コマンド ラインに次のように入力します。

```
<Compiler_Name> options files...
```

ここで、<Compiler_Name> は mb-gcc または arm-xilinx-eabi-gcc です。C++ プログラムをコンパイルするには、mb-g++ または arm-xilinx-eabi-g++ コマンドを使用します。

入力ファイル

コンパイラの入力ファイルとして、次の中から 1 つまたは複数のファイルを指定します。

- C ソース ファイル
- C++ ソース ファイル
- アセンブリ ファイル
- オブジェクト ファイル
- リンカー スクリプト

注記: これらのファイルの指定はオプションです。入力ファイルを指定しない場合は、リンカー (mb-ld または arm-xilinx-eabi-ld) のデフォルト リンカー スクリプトが使用されます。

各ファイルのデフォルトの拡張子については、[表 2-1](#) を参照してください。上記のファイルに加え、コンパイラではライブラリ ファイルである libc.a、libgcc.a、libm.a、および libxil.a が暗示的に参照されます。これらのファイルは、デフォルトで Vivado のインストール ディレクトリにあります。G++ コンパイラを使用した場合は、libsupc++.a および libstdc++.a も参照されます。libsupc++.a は C++ 言語サポート、libstdc++.a は C++ プラットフォーム ライブラリです。

出力ファイル

コンパイラでは、次のファイルが出力として生成されます。

- ELF ファイル。Windows でのデフォルトの出力ファイル名は `a.exe` です。
- アセンブリ ファイル (`-save-temps` または `-S` オプションを使用した場合)
- オブジェクト ファイル (`-save-temps` または `-c` オプションを使用した場合)
- プリプロセッサ出力ファイル (`-save-temps` オプションを使用した場合、`.i` または `.ii` ファイル)

ファイル タイプとその拡張子

GNU コンパイラはファイルの拡張子からファイル タイプを判断します。表 2-1 に、有効な拡張子およびそれに対応するファイル タイプを示します。GCC ラッパーにより、ファイル タイプに応じた適切なツールが呼び出されます。

表 2-1: ファイルの拡張子

拡張子	ファイル タイプ
<code>.c</code>	C ファイル
<code>.C</code>	C++ ファイル
<code>.cxx</code>	C++ ファイル
<code>.cpp</code>	C++ ファイル
<code>.c++</code>	C++ ファイル
<code>.cc</code>	C++ ファイル
<code>.S</code>	アセンブリ ファイル (プリプロセッサ指示子を含む場合もある)
<code>.s</code>	アセンブリ ファイル (プリプロセッサ指示子は含まない)

ライブラリ

表 2-2 に、`mb_gcc` および `arm-xilinx-eabi-gcc` コンパイラに必要なライブラリをリストします。

表 2-2: コンパイラで使用されるライブラリ

ライブラリ	説明
<code>libxil.a</code>	Vivado ツール用に開発されたドライバー、ソフトウェア サービス (XiMFS など)、初期化ファイルを含む
<code>libc.a</code>	<code>strcmp</code> 、 <code>strlen</code> などの関数を含む標準 C ライブラリ
<code>libgcc.a</code>	浮動小数点および 64 ビット演算用のエミュレーション ルーチンを含む GCC の下位ライブラリ
<code>libm.a</code>	<code>cos</code> 、 <code>sine</code> などの関数を含む数学ライブラリ
<code>libsupc++.a</code>	例外処理、RTTI などのルーチンを含む C++ サポート ライブラリ
<code>libstdc++.a</code>	C++ 標準プラットフォーム ライブラリ。ストリーム I/O、ファイル I/O、文字列処理などの標準言語クラスを含む

ライブラリはどちらのコンパイラでも自動的にリンクされます。標準ライブラリとは別のライブラリを使用する場合は、使用するライブラリの検索パスを指定する必要があります。`libxil.a` は、ドライバーおよびライブラリのルーチンを追加するため、変更されます。

言語タイプ

GCC コンパイラは、C および C++ の両言語を認識し、それに合わせたコードを生成します。GCC の規則により、ソース ファイルに GCC または G++ コンパイラを同様に使用することが可能です。使用するコンパイラとソース ファイルの拡張子に基づいて、入力ファイルおよび出力ファイルの言語が決まります。

GCC コンパイラを使用する場合、プログラムの言語は [13 ページの表 2-1](#) に示すようにファイルの拡張子で決まります。ファイル拡張子から判断して C++ ソース ファイルであると識別された場合は、言語は C++ に設定されます。つまり、C コードが CC ファイルに含まれていると、GCC コンパイラを使用した場合でも、関数名のマングル処理が行われます。

GCC と G++ の主な違いは、G++ ではデフォルト言語がファイルの拡張子にかかわらず C++ に自動的に設定され、C++ サポート ライブラリが読み込まれる点です。そのため、C ファイル (.c) に含まれる C コードを G++ でコンパイルすると、関数名のマングル処理が行われます。

名前マングル処理は、C++ などシンボルのオーバーロードをサポートする言語に特有の概念です。渡された引数によって異なる処理を実行し、異なる戻り値を返すことができる関数を、オーバーロード (多重定義) 関数と呼びます。これをサポートするため、C++ コンパイラではその関数名で呼び出される関数のタイプをエンコードして、同じ名前の関数に複数の定義が存在しないようにします。

一部のソース ファイルに C コードが含まれ、その他に C++ コードが含まれる混合コンパイル モードを使用する (一部のファイルのコンパイルに GCC を使用し、その他のファイルのコンパイルに G++ を使用する) 場合は、名前マングル処理に注意する必要があります。C シンボルに対して名前マングル処理が行われないようにするには、シンボル宣言に次の文を使用します。

```
#ifdef __cplusplus
extern "C" {
#endif

int foo();
int morefoo();

#ifdef __cplusplus
}
#endif
```

これらの宣言がヘッダー ファイルで使用されるようにし、ソース ファイルすべてに適用されるようにします。これにより、これらのシンボルの定義および参照をコンパイルする際、C 言語が使用されるようになります。

注記: Vivado のすべてのドライバーおよびライブラリは、すべてのヘッダー ファイルで上記の規則に従います。G++ を使用してコンパイルする場合は、各ドライバーおよびライブラリに記述されているように、必要なヘッダー ファイルを含める必要があります。これにより、コンパイラでライブラリ シンボルが C タイプであることが確実に認識されます。

どちらのコンパイラでコンパイルする場合でも、ファイルを特定の言語に指定するには `-x lang` オプションを使用します。このオプションの詳細は、GNU のウェブサイトの GCC マニュアルを参照してください。[付録 B 「その他のリソースおよび法的通知」](#) に、このマニュアルへのリンクがあります。

- GCC コンパイラを使用する場合は、`libstdc++.a` および `libsupc++.a` は自動的にリンクされません。
- C++ プログラムをコンパイルするときは、G++ タイプのコンパイラを使用し、必要なサポート ライブラリがすべて自動的にリンクされるようにします。
- また、GCC コマンドに `-lstdc++` および `-lsupc++` を追加することも可能です。

それぞれの言語に応じてコンパイルを起動する方法は、GNU のオンライン マニュアルを参照してください。

よく使用されるコンパイラ オプション: 早見表

MicroBlaze および Arm の両方のコンパイラで共通のコンパイラ オプションが次の表にわかりやすくまとめられています。

注記: これらのオプションでは、大文字と小文字が区別されます。

オプション名をクリックすると、そのオプションの説明にジャンプします。

一般オプション		ライブラリ検索オプション
-E	-Wp,option	-l libraryname
-S	-Wa,option	-L Lib Directory
-c	-Wl,option	
-g	-help	ヘッダー ファイル検索オプション
-gstabs	-B directory	-I Directory Name
-On	-L directory	
-v	-I directory	リンカー オプション
-save-temps	-l library	-defsym _STACK_SIZE=value
-o filename		-defsym _HEAP_SIZE=value

一般オプション

-E

プリプロセスのみを実行し、コンパイル、アセンブリ、リンクは実行しません。プリプロセスの結果は、標準の出力デバイスに表示されます。

-S

コンパイルのみを実行し、アセンブリ、リンクは実行しません。 .s ファイルを生成します。

-C

コンパイルおよびアセンブリのみを実行し、リンクは実行しません。 .o ファイルを生成します。

-g

出力ファイルに DWARF2 ベースのデバッグ情報を追加します。このデバッグ情報は、GNU デバッガー (mb-gdb または arm-xilinx-eabi-gdb) により必要とされます。デバッガーでは、ソース レベルまたはアセンブリ レベルでデバッグを実行できます。このオプションは、入力が C または C++ ソース ファイルである場合にのみデバッグ情報を追加します。

-gstabs

ソース レベルのアセンブリ ファイル (.S) およびアセンブリ ファイル シンボルに STABS ベースのデバッグ情報を追加します。これはアセンブラー オプションで、GNU アセンブラー (mb-as または arm-xilinx-eabi-as) に直接渡されます。アセンブリ ファイルがコンパイラ (mb-gcc または arm-xilinx-eabi-gcc) を使用してコンパイルされている場合は、-Wa を前に付けてください。

-On

GNU コンパイラの最適化レベルを指定します。次の表にある最適化レベルは、C および C++ ソース ファイルにのみ適用されます。

表 2-3: 最適化レベル

n	最適化
0	最適化は実行されません。
1	中レベルの最適化が実行されます。
2	完全な最適化を実行します。
3	完全な最適化を実行します。サブプログラムをインライン化します。
S	サイズを小さくするよう最適化します。

注記: 最適化レベルを 1 以上にすると、コードの構成が変わります。コードのデバッグ中は、最適化レベルを 0 にすることをお勧めします。最適化したプログラムを GDB でデバッグすると、結果が不一致のように見える場合があります。

-V

コンパイラおよびコンパイルに関連するすべてのツールを詳細モードで実行します。このオプションを使用すると、ツールで使用されたオプションの詳細が得られるので、各ツールのデフォルト オプションを見つけやすくなります。

-save-temps

コンパイル中に生成された中間ファイルを保存します。次のファイルが保存されます。

- プリプロセッサ出力 (C コードでは `input_file_name.i`、C++ では `input_file_name.ii`)
- アセンブリ フォーマットのコンパイラ (cc1) 出力 (`input_file_name.s`)
- ELF フォーマットのアセンブラー出力 (`input_file_name.s`)

デフォルトでは、コンパイルの内容全体が `a.out` に保存されます。

-o *filename*

コンパイルの出力ファイルは、デフォルトでは `a.out` という ELF ファイルです。このファイル名は、`-o` オプションを使用して変更できます。出力ファイルは ELF フォーマットで生成されます。

-Wp, *option*

-Wa, *option*

-Wl, *option*

コンパイラ (`mb-gcc` または `arm-xilinx-eabi-gcc`) はラッパー ファイルで、プリプロセッサ、コンパイラ (`cc1`)、アセンブラー、リンカーなど呼び出します。これらのツールは、最上位コンパイラを介してまとめて、または個別に実行できます。

これらのツールでは必要でも、最上位のコンパイラでは不要なオプションもあります。そのようなオプションは、次の表に示すように指定します。

表 2-4: ツール別オプションの指定方法

オプション	ツール	例
<code>-Wp,option</code>	プリプロセッサ	<code>mb-gcc -Wp,-D -Wp, MYDEFINE ...</code> プリプロセッサで <code>-D MYDEFINE</code> オプションを使用してシンボル <code>MYDEFINE</code> を定義します。
<code>-Wa,option</code>	アセンブラー	<code>mb-as -Wa, ...</code> アセンブラーで MicroBlaze プロセッサをターゲットに指定します。
<code>-Wl,option</code>	リンカー	<code>mb-gcc -Wl,-M ...</code> <code>-M</code> オプションを使用してマップ ファイルを生成します。

`-help`

GNU コンパイラで使用可能なオプションに関する情報を表示します。GCC のマニュアルも参照してください。

`-B directory`

C のランタイムのライブラリ検索パスに `directory` を追加します。

`-L directory`

ライブラリ検索パスに `directory` を追加します。

`-I directory`

ヘッダー検索パスに `directory` を追加します。

`-l library`

未定義のシンボルを `library` で検索します。

注記: このコマンド ライン オプションで指定したライブラリ名に、`lib` という接頭辞が追加されます。

ライブラリ検索オプション

`-l libraryname`

デフォルトでは、`libc`、`libm`、`libxil` などの標準ライブラリのみが検索されます。カスタム ライブラリも作成できます。このオプションを使用してそのカスタム ライブラリを指定できます。このオプションで指定したライブラリ名に `lib` という接頭辞が追加されます。

コマンド ラインで指定するオプションの順序には決まりがあり、特に `-l` オプションの位置は重要です。このオプションは、ソース ファイルの後に使用してください。

たとえば、`libproject.a` というカスタム ライブラリを作成している場合は、次のコマンドを使用するとこのライブラリに含まれる関数を使用できます。

```
Compiler Source_Files -L${LIBDIR} -l project
```



注意: `-l library_name` オプションをソース ファイルの前に使用すると、コンパイラでソース ファイルから呼び出される関数を検索できません。これは、コンパイラでの検索は一方向にのみ実行され、使用可能なライブラリのリストは保持されないからです。

-L Lib Directory

ライブラリを検索するディレクトリを指定します。デフォルトのライブラリが検索パスとして設定されており、標準ライブラリはここから検索されますが、`-L` オプションを指定すると、コンパイラの検索パスにライブラリを検索するディレクトリを追加できます。

ヘッダー ファイル検索オプション

-I Directory Name

ヘッダー ファイルを標準パスで検索する前に、`/<dir_name>` で指定したディレクトリで検索するよう指定します。

デフォルトの検索パス

コンパイラ (`mb-gcc` および `arm-xilinx-eabi-gcc`) は、特定のパスからライブラリおよびヘッダー ファイルを検索します。プラットフォーム別の検索パスを、次に示します。

ライブラリ検索順序

ライブラリは、次の順に検索されます。

1. `-L <dir_name>` オプションで指定されたディレクトリ
2. `-B <dir_name>` オプションで指定されたディレクトリ
3. 次のライブラリ
 - a. `${XILINX_}/gnu/processor/platform/processor-lib/lib`
 - b. `${XILINX_}/lib/processor`

注記: `processor` は、MicroBlaze の場合は「`microblaze`」、Arm の場合は「`arm-xilinx-eabi`」です。

ヘッダー ファイル検索順序

ヘッダー ファイルは、次の順に検索されます。

1. `-I <dir_name>` で指定されたディレクトリ
2. 次のヘッダー ファイル
 - a. `${XILINX_}/gnu/processor/platform/lib/gcc/processor/{gcc version}/include`
 - b. `${XILINX_}/gnu/processor/platform/processor-lib/include`

初期化ファイル検索順序

初期化ファイルは、次の順に検索されます。

1. `-B <dir_name>` で指定されたディレクトリ
2. `${XILINX_}/gnu/processor/platform/processor-lib/lib`
3. 次のライブラリ
 - a. `$XILINX_/gnu/<processor>/platform/<processor-lib>/lib`
 - b. `$XILINX_/lib/processor`

説明:

- 。 `<processor>` は、MicroBlaze プロセッサの場合は「microblaze」、Arm プロセッサの場合は「arm-xilinx-eabi」です。
- 。 `<processor-lib>` は、MicroBlaze プロセッサの場合は「microblaze-xilinx-elf」、Arm プロセッサの場合は「arm-xilinx-eabi」です。

注記: `<platform>` は、Linux の場合は「lin」、Linux 64 ビットの場合は「lin64」、Windows Cygwin の場合は「nt」です。

リンカー オプション

`-defsym _STACK_SIZE=value`

スタック領域に割り当てられているメモリ容量を変更します。変数 `_STACK_SIZE` は、スタック領域に割り当てられている総容量を示します。デフォルト値は 100 ワード (400 バイト) です。スタック領域とヒープ領域の合計に 400 バイト以上必要な場合は、このオプションを使用して `_STACK_SIZE` の値を大きくします。値はバイト単位で指定します。

プログラムでスタック領域を大きくする必要がある場合があります。プログラムで必要なスタック サイズが割り当てられているサイズよりも大きい場合は、ほかのプログラムの不正な領域に書き込みが行われ、コードが正常に実行されない場合があります。

注記: ザイリンクスが提供する C ランタイム (CRT) ファイルにリンクされたプログラムでは、スタック サイズを 16 バイト (0x0010) 以上にする必要があります。

`-defsym _HEAP_SIZE=value`

ヒープ領域に割り当てられているメモリ容量を変更します。変数 `_HEAP_SIZE` のデフォルト値は 0 です。

ダイナミック メモリ割り当てルーチンは、ヒープ領域を使用します。プログラムでこのようにヒープ領域を使用する場合は、`_HEAP_SIZE` に適切な値を設定する必要があります。

上級ユーザーであれば、リンカー スクリプトを IP インテグレーターから直接生成することもできます。

メモリのレイアウト

MicroBlaze および Arm プロセッサは、32 ビットの論理アドレスを使用し、システムのメモリに 0x0 から 0xFFFFFFFF までのアドレスを指定できます。この範囲は、予約済みメモリと I/O メモリに分けることができます。

予約済みメモリ

予約済みメモリは、ハードウェアおよびソフトウェアのプログラム環境で専用を使用するため定義される領域で、割り込みベクター ロケーションおよび OS レベルのルーチンを含むメモリが一般的に予約済みメモリとなります。[表 2-5](#) には、MicroBlaze および Arm プロセッサそれぞれに対して規定されている予約済みメモリ ロケーションがまとめられています。これらのメモリ ロケーションの詳細は、使用するプロセッサのリファレンス マニュアルを参照してください。

Arm メモリ マップについては『Zynq-7000 SoC テクニカル リファレンス マニュアル』(UG585) [\[参照 2\]](#) を参照してください。

注記: ハードウェアだけでなく、ソフトウェア環境に対してもメモリが予約される場合があります。ソフトウェアに対しメモリ ロケーションが予約されているかどうかを確認するには、使用するソフトウェア プラットフォームのマニュアルを参照してください。

表 2-5: ハードウェアで予約されるメモリ ロケーション

プロセッサ	予約済みメモリ	用途	デフォルトのテキスト開始アドレス
MicroBlaze	0x0 ~ 0x4F	リセット、割り込み、例外、その他の予約済みベクターの場所	0x50
Cortex A9 Arm			

I/O メモリ

I/O メモリは、プログラムがプロセッサ バス上のメモリ マップされたペリフェラルと通信するために使用されます。これらのアドレスは、ハードウェア プラットフォーム仕様の一部として定義されます。

ユーザーおよびプログラム メモリ

ユーザーおよびプログラム メモリとは、コンパイルされた実行ファイルの実行に必要なメモリすべてを指します。命令、読み出し専用データ、読み出し/書き込みデータ、プログラム スタック、プログラム ヒープの保存に使用されます。これらのセクションは、システム内のアドレス指定可能なメモリであればどこにでも保存できます。デフォルトでは、コンパイラで生成されたコードおよびデータは、[表 2-5](#) にリストされているアドレスから開始して、そこから連続するアドレスのメモリ ロケーションに保存されます。これが最も一般的なプログラムのメモリ レイアウトです。プログラムの開始位置を変更するには、リンカーで MicroBlaze の場合は `_TEXT_START_ADDR` シンボル、Arm の場合は `_START_ADDR` シンボルを定義します。

ELF ファイルのセクションをそれぞれ異なるメモリに分割する必要がある場合は、リンカー コマンド 言語を使用します。詳細は、[24 ページの「リンカー スクリプト」](#) を参照してください。実行ファイルのメモリ マップを変更するのは、次のような場合です。

- 。 長いコードを複数の小型メモリに分割する場合
- 。 頻繁に実行されるセクションを高速メモリにマップする場合
- 。 読み出し専用セグメントを不揮発性のフラッシュ メモリにマップする場合

実行ファイルの分割方法に制限はありません。分割は、出力セクション レベルまたは関数レベル、データ レベルで行うことができます。生成される ELF ファイルが連続しておらず、メモリ マップにギャップが存在することがあります。予約済みロケーションを使用しないよう注意してください。

または、ツールで提供される予約済みメモリ ロケーションのデフォルト バイナリ データを変更することも可能です。この場合、リンカーで提供されるデフォルトのスタートアップ ファイルおよびメモリ マップを置き換える必要があります。

オブジェクト ファイルのセクション

実行ファイルは、オブジェクト ファイル (.o ファイル) の入力セクションをリンクして作成します。デフォルトでは、標準であり明確に定義されたセクションからコンパイラがコードを作成します。各セクションには、その意味および目的に応じて名前が付けられています。オブジェクト ファイルのさまざまな標準セクションを次に示します。

これらのセクションに加え、カスタム セクションを作成して、メモリに割り当てることもできます。

Sectional Layout of an object or an Executable File

.text	Text Section
.rodata	Read-Only Data Section
.sdata2	Small Read-Only Data Section
.sbss2	Small Read-Only Uninitialized Data Section
.data	Read-Write Data Section
.sdata	Small Read-Write Data Section
.sbss	Small Uninitialized Data Section
.bss	Uninitialized Data Section
.heap	Program Heap Memory Section
.stack	Program Stack Memory Section

X11005

図 2-2: オブジェクト (実行) ファイルのセクション レイアウト

.init、.fini、.ctors、.dtors、.got、.got2、.eh_frame など、通常は変更しない予約済みセクションがあります。

`.text`

オブジェクト ファイルのこのセクションには実行可能なプログラム命令が含まれており、`x` (実行コード)、`r` (読み出し専用)、および `i` (初期化) フラグが付けられています。プロセッサ命令バスでアドレス指定可能な初期化済み ROM に割り当てることができます。

`.rodata`

このセクションには読み出し専用データが含まれており、`r` (読み出し専用) および `i` (初期化) フラグが付けられています。`.text` セクションと同様、プロセッサ命令バスでアドレス指定可能な初期化済み ROM に割り当てることができます。

`.sdata2`

`.rodata` セクションと同様ですが、8 バイト未満のサイズの小さい読み出し専用データが含まれます。このセクションのデータには、すべて読み出し専用のスモール データ アンカーへの参照を使用してアクセスします。これにより、このセクションのすべてのデータに 1 つの命令でアクセスできます。このセクションに配置するデータのサイズは、`-G` オプションで変更できます。`r` (読み出し専用) および `i` (初期化) フラグが付けられています。

`.data`

このセクションには読み出し/書き込みデータが含まれており、`w` (読み出し/書き込み) および `i` (初期化) フラグが付けられています。初期化済みの RAM にマップする必要があります。ROM にはマップできません。

`.sdata`

このセクションには、8 バイトよりも小さい読み出し/書き込み可能なデータが含まれます。このセクションに配置するデータのサイズは、`-G` オプションで変更できます。このセクションのデータには、すべて読み出し/書き込みのスモール データ アンカーへの参照を使用してアクセスします。これにより、このセクションのすべてのデータに 1 つの命令でアクセスできます。`w` (読み出し/書き込み) および `i` (初期化) フラグが付けられており、初期化済み RAM にマップする必要があります。

`.sbss2`

このセクションには、8 バイトよりも小さい読み出し専用の初期化されないデータが含まれます。このセクションに配置するデータのサイズは、`-G` オプションで変更できます。`r` (読み出し) フラグが付けられており、ROM にマップする必要があります。

`.sbss`

このセクションには、8 バイトよりも小さい初期化されないデータが含まれます。このセクションに配置するデータのサイズは、`-G` オプションで変更できます。このセクションには、`w` (読み出し/書き込み) フラグが付けられており、RAM にマップする必要があります。

`.bss`

このセクションには、初期化されていないデータが含まれます。`w` (読み出し/書き込み) フラグが付けられており、RAM にマップする必要があります。

`.heap`

このセクションには、グローバルプログラム ヒープとして使用される初期化されていないデータが含まれます。このセクションのメモリは、ダイナミック メモリ割り当てルーチンにより割り当てられます。RAM にマップする必要があります。

`.stack`

このセクションには、プログラム スタックとして使用される初期化されていないデータが含まれます。RAM にマップする必要があります。通常は、`.heap` セクションのすぐ後に配置されます。リンカーによっては、`.stack` と `.heap` セクションが `.bss_stack` というセクションにマージされているように見えます。

`.init`

このセクションには言語初期化コードが含まれており、`.text` セクションと同じフラグが付けられています。初期化済みの ROM にマップする必要があります。

`.fini`

このセクションには言語クリーンアップコードが含まれており、`.text` セクションと同じフラグが付けられています。初期化済みの ROM にマップする必要があります。

`.ctors`

このセクションにはプログラムの起動時に呼び出す必要のある関数がリストされており、`.data` と同じフラグが付けられています。初期化済みの RAM にマップする必要があります。

`.dtors`

このセクションにはプログラムの終了時に呼び出す必要のある関数がリストされており、`.data` と同じフラグが付けられています。初期化済みの RAM にマップする必要があります。

`.got2/.got`

このセクションにはプログラム データへのポインターが含まれており、`.data` と同じフラグが付けられています。初期化済みの RAM にマップする必要があります。

`.eh_frame`

このセクションには、例外処理用のフレーム巻き戻し情報が含まれます。このセクションには、`.rodata` と同じフラグが付けられており、初期化済みの ROM にマップできます。

`.tbss`

このセクションには、初期化されないスレッドのローカル データが含まれており、このデータはプログラムのメモリ イメージの一部になります。このセクションには、`.bss` と同じフラグが付けられており、RAM にマップする必要があります。

`.tdata`

このセクションには、初期化されたスレッドのローカル データが含まれており、このデータはプログラムのメモリ イメージの一部になります。初期化済みの RAM にマップする必要があります。

.gcc_except_table

このセクションには、言語特定のデータが含まれています。初期化済みの RAM にマップする必要があります。

.jcr

このセクションには、コンパイルされた Java クラスを登録するために必要な情報が含まれています。内容はコンパイラに特化したもので、コンパイラ初期化関数が使用します。初期化済みの RAM にマップする必要があります。

.fixup

このセクションには、フィックスアップ ページテーブルやフィックスアップ レコード テーブルを実行するために必要な情報が含まれています。初期化済みの RAM にマップする必要があります。

リンカー スクリプト

リンカー ユーティリティは、リンカー スクリプトで指定したコマンドを使用してユーザー プログラムを複数のメモリ ブロックに分割します。リンカー スクリプトは、すべての入力オブジェクト ファイルのすべてのセクションから実行ファイルへのマップを記述します。出力セクションは、システムのメモリにマップされます。プログラム データを連続するメモリに割り当てるというデフォルト操作を変更しない場合は、リンカー スクリプトは必要ありません。リンカーには、セクションの内容を連続的に割り当てる、デフォルトのリンカー スクリプトが用意されています。

プログラムの開始ロケーションのみを変更するには、次の例に示すように、MicroBlaze の場合は `_TEXT_START_ADDR` シンボル、Arm の場合は `START_ADDR` シンボルを定義します。

```
mb-gcc <input files and flags> -Wl,-defsym -Wl,_TEXT_START_ADDR=0x100
mb-ld <.o files> -defsym _TEXT_START_ADDR=0x100
```

`$XILINX_/gnu/<procname>/<platform>/<processor_name>/lib/ldscripts` にある、リンカーで 사용되는デフォルト スクリプトには次のものがあります。

- `elf32<procname>.x`: この後に示すオプションのいずれも使用されていない場合のデフォルト
- `elf32<procname>.xn`: `-n` オプションを使用した場合
- `elf32<procname>.xbn`: `-N` オプションを使用した場合
- `elf32<procname>.xr`: `-r` オプションを使用した場合
- `elf32<procname>.xu`: `-Ur` オプションを使用した場合

ここで、`<procname>` は `microblaze`、`<processor_name>` は `microblaze`、`<platform>` は `lin` または `nt` です。

リンカー スクリプトを使用するには、GCC コマンド ラインで指定します。次のコマンド ライン オプションを使用してください。コンパイラには、次のように `-T <script>` オプションを使用します。

```
compiler -T <linker_script> <Other Options and Input Files>
```

リンカーを個別に実行する場合は、リンカー スクリプトを次のように指定します。

```
compiler -T <linker_script> <Other Options and Input Files>
```

このコマンドを使用すると、デフォルトのリンカー スクリプトの代わりに指定したリンカー スクリプトが使用されます。プログラム用のリンカー スクリプトは、IP インテグレーターおよび SDK から生成できます。

IP インテグレーターまたは SDK で、[Tools] → [Generate Linker Script] をクリックします。

これでリンカー スクリプト生成ユーティリティが開きます。セクションからメモリへのマップは、ここで行います。スタックおよびヒープのサイズとメモリ マップもここで設定できます。リンカー スクリプトが生成されると、IP インテグレーターまたは SDK 内で対応するアプリケーションをコンパイルしたときにそのスクリプトが自動的に GCC に入力されます。

リンカー スクリプトは、メモリに変数または関数を割り当てるために使用できます。これには、C コードのセクション属性を使用します。また、リンカー スクリプトでメモリのセクションにオブジェクト ファイルを割り当てることもできます。これらの機能およびその他の機能については、オンライン `binutils` マニュアルの一部である GNU リンカーのマニュアルを参照してください。GNU マニュアルへのリンクは[付録 B 「その他のリソースおよび法的通知」](#)にあります。MicroBlaze プロセッサのリンカー スクリプトにより割り当てられる入力セクションの詳細については、[33 ページの「MicroBlaze リンカー スクリプトで割り当てられるセクション」](#)を参照してください。

MicroBlaze コンパイラの使用法とオプション

MicroBlaze 用の GNU コンパイラは、標準の GNU ソースに基づいています。MicroBlaze コンパイラに特定の機能およびオプションを次に説明します。MicroBlaze コンパイラでコンパイルする場合、プリプロセッサで自動的に `__MICROBLAZE__` 定義が使用されます。この定義は、どのような条件コードでも使用できます。

MicroBlaze コンパイラ

ザイリンクス MicroBlaze ソフト プロセッサ用の `mb-gcc` コンパイラでは、専用のオプションが追加されているだけでなく、GNU コンパイラでサポートされている一部のオプションが変更されています。ここでは、その両方のオプションについて説明します。

MicroBlaze コンパイラ オプション: 早見表

オプション名をクリックすると、そのオプションの説明にジャンプします。

プロセッサ機能選択オプション	一般プログラム オプション
-mcpu=vX.YY.Z	-msmall-divides
-mlittle-endian / -mbig-endian	-mxl-gp-opt
-mno-xl-soft-mul	-mno-clearbss
-mxl-multiply-high	-mxl-stack-check
-mno-xl-multiply-high	アプリケーション実行モード
-mxl-soft-mul	-xl-mode-executable
-mxl-barrel-shift	-xl-mode-bootstrap
-mno-xl-barrel-shift	-xl-mode-novectors
-mxl-pattern-compare	MicroBlaze リンカー オプション
-mno-xl-pattern-compare	-defsym _TEXT_START_ADDR=value
-mhard-float	-relax
-msoft-float	-N
-mxl-float-convert	
-mxl-float-sqrt	

プロセッサ機能選択オプション

`-mcpu=vX.YY.Z`

MicroBlaze ハードウェアのバージョン `v.X.YY.Z` に適したコードを生成します。プロセッサ用に最適化された正しいコードを生成するには、このオプションでプロセッサのハードウェア バージョンを指定します。

指定するバージョンによって、`-mcpu` の動作は異なります。

- `Pr-v3.00.a`: 3 段プロセッサ パイプライン モードを使用します。命令を遅延スロットに移動する例外は禁止されません。
- `v3.00.a` and `v4.00.a`: 3 段プロセッサ パイプライン モードを使用します。命令を遅延スロットに移動する例外は禁止されます。
- `v5.00.a` 以降: 5 段プロセッサ パイプライン モードを使用します。命令を遅延スロットに移動する例外は禁止されません。

`-mlittle-endian` / `-mbig-endian`

コンパイルされているコードのターゲット マシンのエンディアンを選択するため、これらのオプションを使用します。生成されるバイナリ オブジェクト ファイルのエンディアンもこのオプションに基づいて設定されます。下位ツール (`as`、`cc1`、`cc1plus`、`ld` など) で該当するエンディアンを設定するため、GCC ドライバーはオプションを下位ツールに渡します。

デフォルトは `-mbig-endian` です。

注記: 複数のエンディアンをとりまぜたオブジェクト ファイルをリンクさせることはできません。

`-mno-xl-soft-mul`

32 ビット乗算に対し、ハードウェア乗算命令を使用できるようにします。

MicroBlaze プロセッサには、ハードウェア乗算リソースの使用をオン/オフにするオプションがあります。

MicroBlaze でハードウェア乗算オプションがオンになっている場合は、このオプションを使用する必要があります。ハードウェア乗算を使用すると、アプリケーションのパフォーマンスが向上します。このオプションを使用すると、C プリプロセッサ定義 `HAVE_HW_MUL` が自動的に定義されます。これにより、この機能が使用可能かどうかに基づいて、ハードウェアに適した C またはアセンブリ コードが記述されます。MicroBlaze での乗算器オプションの使用については、『MicroBlaze プロセッサ リファレンス ガイド』(UG081) [参照 3] を参照してください。

`-mxl-multiply-high`

MicroBlaze には、32X32 ビット乗算の上位 32 ビットを計算する命令をイネーブルにするオプションがあります。このオプションは、コンパイラでこれらの上位ビット乗算命令を使用するよう指示します。このオプションを使用すると、C プリプロセッサ定義 `HAVE_HW_MUL_HIGH` が自動的に定義されます。これにより、この機能が使用可能かどうかに基づいて、ハードウェアに適した C またはアセンブリ コードが記述されます。MicroBlaze での上位ビット乗算命令の使用については、『MicroBlaze プロセッサ リファレンス ガイド』(UG081) [参照 3] を参照してください。

`-mno-xl-multiply-high`

上位ビット乗算命令を使用しないよう指定します。このオプションがデフォルトです。

`-mxl-soft-mul`

MicroBlaze にハードウェア乗算器がないことを示します。デバイスにハードウェア乗算器がない場合、32 ビット乗算操作はソフトウェア エミュレーション ルーチン `__mulsi3` に置換されます。このオプションがデフォルトです。

`-mno-xl-soft-div`

MicroBlaze にハードウェア除算器をインスタンス化できます。除算器がある場合、このオプションを指定すると、コンパイルされているプログラムでハードウェア除算命令が使用できるようになります。

プログラムに除算処理が多数含まれている場合、このオプションを使用するとパフォーマンスが向上します。このオプションを使用すると、C プリプロセッサ定義 `HAVE_HW_DIV` が自動的に定義されます。これにより、この機能が使用可能かどうかに基づいて、ハードウェアに適した C またはアセンブリ コードが記述されます。MicroBlaze でのハードウェア除算器オプションの使用については、『MicroBlaze プロセッサ リファレンス ガイド』(UG081) [参照 3] を参照してください。

`-mxl-soft-div`

ターゲットの MicroBlaze にハードウェア除算器がないことを示します。

このオプションがデフォルトです。このオプションを設定すると、すべての 32 ビット除算が対応するソフトウェア エミュレーション ルーチン (`__divsi3`, `__udivsi3`) に置換されます。

`-mxl-barrel-shift`

MicroBlaze プロセッサは、パレル シフターを組み込むようコンフィギュレーションできます。プロセッサのパレル シフト機能を使用するには、`-mxl-barrel-shift` オプションを使用します。

デフォルトではパレル シフターはないと判断され、オペランドのシフトには加算と乗算が使用されます。パレル シフターをイネーブルにすると、特に浮動小数点ライブラリを使用している場合に、アプリケーションの速度が大幅に向上します。このオプションを使用すると、C プリプロセッサ定義 `HAVE_HW_BSHIFT` が自動的に定義されます。これにより、この機能が使用可能かどうかに基づいて、ハードウェアに適した C またはアセンブリ コードが記述されます。MicroBlaze でのパレル シフターのオプションの使用については、『MicroBlaze プロセッサ リファレンス ガイド』(UG081) [参照 3] を参照してください。

`-mno-xl-barrel-shift`

ハードウェア パレルシフト命令を使用しないよう指定します。このオプションがデフォルトです。

`-mxl-pattern-compare`

コンパイラでのパターン比較命令の使用をオンにします。

パターン比較命令を使用すると、プログラムのブール演算の速度が向上します。また、パターン比較操作では、`strcpy`、`strlen`、`strcmp` などの文字列処理ルーチンにおいて、バイト長ではなくワード長での操作が可能になります。文字列処理ルーチンを多用するプログラムでは、これにより処理速度が大幅に向上します。このオプションを使用すると、C プリプロセッサ定義 `HAVE_HW_PCMP` が自動的に定義されます。これにより、この機能が使用可能かどうかに基づいて、ハードウェアに適した C またはアセンブリ コードが記述されます。MicroBlaze でのパターン比較オプションの使用については、『MicroBlaze プロセッサ リファレンス ガイド』(UG081) [参照 3] を参照してください。

`-mno-xl-pattern-compare`

パターン比較命令を使用しないよう指定します。これがデフォルトです。

`-mhard-float`

コンパイラでの単精度浮動小数点命令 (`fadd`、`frsub`、`fmul`、`fdiv`) の使用をオンにします。

また、`fcmp.p` 命令 (`p` は `le`、`ge`、`lt`、`gt`、`eq`、`ne` などの述語条件) も使用します。これらの命令は、ハードウェアで FPU がイネーブルの場合に、MicroBlaze でデコードおよび実行されます。このオプションを使用すると、C プリプロセッサ定義 `HAVE_HW_FPU` が自動的に定義されます。これにより、この機能が使用可能かどうかに基づいて、ハードウェアに適した C またはアセンブリ コードが記述されます。MicroBlaze でのハードウェア浮動小数点の使用については、『MicroBlaze プロセッサ リファレンス ガイド』(UG081) [\[参照 3\]](#) を参照してください。

`-msoft-float`

浮動小数点演算のソフトウェア エミュレーションを使用するよう指定します。このオプションがデフォルトです。

`-mxl-float-convert`

コンパイラでの単精度浮動小数点変換命令 (`fint` および `flt`) の使用をオンにします。ハードウェアで FPU がイネーブルになっており、このオプションの命令がイネーブルになっていると、この命令が MicroBlaze によりネイティブにデコードされ、実行されます。

MicroBlaze でのハードウェア浮動小数点の使用については、『MicroBlaze プロセッサ リファレンス マニュアル』(UG081) [\[参照 3\]](#) を参照してください。

`-mxl-float-sqrt`

コンパイラでの単精度浮動小数点平方根命令 (`fsqrt`) の使用をオンにします。ハードウェアで FPU がイネーブルになっており、このオプションの命令がイネーブルになっていると、この命令が MicroBlaze によりネイティブにデコードされ、実行されます。

MicroBlaze でのハードウェア浮動小数点の使用については、『MicroBlaze プロセッサ リファレンス マニュアル』(UG081) [\[参照 3\]](#) を参照してください。

一般プログラム オプション

-msmall-divides

ハードウェア除算器がない場合に、小さい数値の除算に対して最適化されたコードを生成します。分母と分子が 0 ~ 15 の間にあるような符号付き整数の除算では、このオプションを使用するとルックアップ テーブルに基づく高速の除算ができるようになります。ハードウェア除算器がイネーブルの場合は、このオプションは無視されます。

-mx1-gp-opt

プログラムに上位 16 ビットに 0 以外の値が含まれているアドレスがある場合、読み込みまたは格納操作には 2 つの命令が必要です。

MicroBlaze ABI には 2 つのグローバル スモール データ領域があり、それぞれ 64KB までのデータを保存できます。これらのデータ領域にあるメモリ ロケーションには、スモール データ領域アンカーおよび 16 ビットの即値を使用してアクセスできるので、スモール データ領域への読み込みまたは格納操作を 1 つの命令のみで実行できます。この最適化をオンにするには、`-mx1-gp-opt` オプションを使用します。サイズがしきい値未満の変数はこれらの領域に保存され、少ない命令数でアドレス指定できます。アドレスはリンク段階で計算されます。



注意: このオプションを使用する場合、プログラムのビルド プロセスのコンパイル コマンドとリンク コマンドの両方で指定する必要があります。どちらか一方のみで使用すると、コンパイル、リンク、またはランタイム エラーが発生する可能性があります。

-mno-clearbss

このオプションは、シミュレーションで使用するプログラムをコンパイルする際に有益です。

C 言語のルールに基づき、初期化されていないグローバル変数は `.bss` セクションに割り当てられ、プログラム実行開始時の値は常に 0 になります。通常、プログラム実行開始時に、`.bss` セクションが 0 で埋められるように C スタートアップ ファイルをループで実行します。また、コンパイラを最適化した場合も、C コードで 0 に割り当てられたグローバル変数が `.bss` セクションに割り当てられます。

シミュレーション環境では、上記の 2 言語機能は余分なオーバーヘッドとなる場合があります。シミュレータによっては、メモリ全体を自動的に 0 にするものがあります。通常的环境であっても、グローバル変数が開始時に 0 になっていなくても良いような C コードを記述できます。そのような場合にこのオプションが役立ちます。このオプションを使用すると、C スタートアップ ファイルにより `.bss` セクションが 0 に初期化されることはなくなります。また、このオプションは `.bss` セクションに 0 に初期化されたグローバル変数を割り当てず、`.data` セクションに移動します。このオプションにより、アプリケーションの起動時間が短縮される場合があります。このオプションを使用する場合はよく注意してください。また、グローバル変数を 0 に初期化することを前提にしているコードを使用しない、またはシミュレーション プラットフォームが自動的にメモリを 0 に初期化するコードを使用しないようにしてください。

-mxl-stack-check

プログラムの実行中にスタック オーバーフローが発生しているかどうかをチェックするよう指定します。

コンパイラは、各関数のプロローグ コード内に、スタック ポインター値と使用可能なメモリを比較するコードを挿入しますが、スタック ポインターが使用可能なメモリを超えている場合は、プログラムはサブルーチン `_stack_overflow_exit` に飛びます。このサブルーチンは、変数 `_stack_overflow_error` の値を 1 に設定します。

スタック オーバーフロー ハンドラーとして機能する関数 `_stack_overflow_exit` をソース コードに含め、標準のスタック オーバーフロー ハンドラーの代わりとして使用することができます。

アプリケーション実行モード

-xl-mode-executable

`mb-gcc` でプログラムをコンパイルする際のデフォルト モードです。`mb-gcc` を使用する場合は、指定する必要はありません。このオプションを使用すると、スタートアップ ファイル `crt0.o` が使用されます。

-xl-mode-bootstrap

ブートローダーを使用して読み込むアプリケーションをコンパイルする際に使用します。通常ブートローダーは、不揮発性メモリにあり、プロセッサ リセット ベクターにマップされます。標準実行ファイルがこのブートローダーで読み込まれた場合、アプリケーション リセット ベクターがブートローダーのリセット ベクターを上書きします。その場合、プロセッサのリセット時にブートローダーが最初に実行されず (通常は最初に実行されるようになっている)、このアプリケーションの再読み込みおよびその他必要な初期化が行われません。

この状況を回避するため、このコンパイラ オプションを使用する必要があります。このコンパイラ オプションを使用すると、プロセッサのリセット時に、アプリケーションではなくブートローダーが実行されます。

上記とは異なる状況で使用されるアプリケーションでは、このオプションは機能しません。このモードでは、`crt2.o` がスタートアップ ファイルとして使用されます。

-xl-mode-novectors

MicroBlaze ベクターを必要としないアプリケーションをコンパイルする際に使用します。通常は、プロセッサのリセット、割り込み、または例外機能を使用しないスタンドアロン アプリケーションで使用します。このオプションを使用すると、ベクターの命令が含まれないので、コード サイズが小さくなります。このモードでは、`crt3.o` がスタートアップ ファイルとして使用されます。



注意: コマンド ラインで、複数の実行モードを指定しないでください。複数のモードを指定すると、シンボルが複数定義されていることが原因でリンク エラーが発生します。

位置独立コード (PIC)

MicroBlaze 用の GNU コンパイラでは、`-fPIC` および `-fpic` オプションがサポートされています。これらのオプションを使用すると、コンパイラで位置独立コード (PIC) を生成できます。この機能は、共有ライブラリおよび再配置可能実行ファイルをインプリメントするために Linux 上で MicroBlaze に対してのみ使用されます。生成されたコードのデータ アクセスをすべて再配置するにはグローバル オフセット テーブル (GOT) が使用され、共有ライブラリへの関数呼び出しにはプロシージャ リンケージ テーブル (PLT) が使用されます。これは、GNU ベースのプラットフォームで再配置可能コードを生成し、共有ライブラリをダイナミックにリンクする際の標準的な方法です。

MicroBlaze アプリケーション バイナリ インターフェイス

MicroBlaze 用の GNU コンパイラは、『MicroBlaze プロセッサ リファレンス ガイド』(UG081) [参照 3] で定義されている、アプリケーション バイナリ インターフェイス (ABI) を使用します。レジスタおよびスタックの使用法に関する規則、コンパイラで使用される標準メモリ モデルの説明は、ABI の資料を参照してください。

MicroBlaze アセンブラー

ザイリンクス MicroBlaze ソフト プロセッサ用の mb-as アセンブラーでは、標準 GNU コンパイラでサポートされているオプションおよび標準 GNU アセンブラーでサポートされているアセンブラー指示子がサポートされています。

mb-as アセンブラーでは、imm 命令以外の MicroBlaze マシン命令セットの opcode がサポートされています。imm 命令は、大きい即値が使用される場合に生成されます。imm 命令を含むコードを記述するためのアセンブリ言語プログラムは必要ありません。MicroBlaze の命令セットの詳細は、『MicroBlaze プロセッサ リファレンス ガイド』(UG081) [参照 3] を参照してください。

mb-as アセンブラーでは、即値オペランドを使用するすべての MicroBlaze 命令を定数またはラベルとして指定する必要があります。命令に PC 相対オペランドが必要な場合は、mb-as アセンブラーによりそれが算出され、必要に応じて imm 命令に含められます。

たとえば、beqi (Branch Immediate if Equal) 命令には PC 相対オペランドが必要です。

アセンブリ プログラムでは、この命令を次のように使用します。

```
beqi r3, mytargetlabel
```

ここで、mytargetlabel は対象となる命令のラベルです。mb-as アセンブラーにより、命令の即値が mytargetlabel - PC として算出されます。

即値が 16 ビットより大きい場合は、imm 命令が自動的に挿入されます。コンパイル時に mytargetlabel の値が不明な場合は、常に imm 命令が挿入されます。不要な imm 命令を削除するには、relax オプションを使用してください。

同様に、命令で大きな定数のオペランドが必要な場合は、アセンブリ言語プログラムで imm 命令ではなくオペランドをそのまま使用する必要があります。たとえば次のコードでは、レジスタ r3 の内容に定数 200,000 を追加し、結果をレジスタ r4 に保存します。

```
addi r4, r3, 200000
```

mb-as アセンブラーは、このオペランドに imm 命令が必要であると判断し、自動的に挿入します。

mb-as アセンブラーでは、アセンブリのプログラムを簡単にするため、標準の MicroBlaze 命令セットに加えていくつかの擬似 opcode がサポートされています。表 2-6 に、サポートされている擬似 opcode を示します。

表 2-6: GNU アセンブラーでサポートされる擬似 Opcode

擬似 Opcode	説明
nop	操作は実行されません。次の命令に置換されます。or R0, R0, R0
la Rd, Ra, Imm	次の命令に置換されます。addik Rd, Ra, imm; = Rd = Ra + Imm;
not Rd, Ra	次の命令に置換されます。xori Rd, Ra, -1
neg Rd, Ra	次の命令に置換されます。rsub Rd, Ra, R0
sub Rd, Ra, Rb	次の命令に置換されます。rsub Rd, Rb, Ra

MicroBlaze リンカー オプション

MicroBlaze ソフト プロセッサ用の mb-ld リンカーでは、GNU コンパイラでサポートされているオプションに加え、追加のオプションも導入されています。このセクションでは、これらのオプションについて説明します。

-defsym `_TEXT_START_ADDR=value`

デフォルトでは、出力コードのテキスト セクションはベース アドレス 0x0 から開始しますが、このオプションを使用すると、このデフォルトを変更できます。mb-gcc コンパイラの実行時にこのオプションを指定すると、出力コードのテキスト セクションは value で指定したアドレスから開始するようになります。

デフォルトの開始アドレスを使用する場合は、-defsym `_TEXT_START_ADDR` を設定する必要はありません。

これはリンカー オプションであり、リンカーを個別に実行する場合に使用します。リンカーを mb-gcc の一部として実行する場合は、次のオプションを使用してください。

```
-Wl,-defsym -Wl,_TEXT_START_ADDR=value
```

-relax

アセンブラーで生成された不要な imm 命令を削除するリンカー オプションです。アセンブラーでは、即値を算出できない命令があると、imm 命令が生成されます。

ほとんどの場合、imm 命令は必要ありません。-relax オプションを使用すると、リンカーにより不要な imm 命令が削除されます。

このオプションは、リンカーを個別に実行した場合にのみ必要です。リンカーを mb-gcc の一部として実行する場合は、このオプションは自動的に指定されます。

-N

テキストおよびデータ セクションを読み出し/書き込み可能にします。データ セグメントはページ揃えされません。このオプションは、MicroBlaze プログラムにのみ必要です。リンカーを GCC コンパイラの一部として実行する場合は、このオプションが自動的に指定されます。リンカーを個別に実行する場合は、このオプションを指定する必要があります。

このオプションの詳細は、GNU のマニュアルを参照してください。

MicroBlaze リンカーでは、セクションをメモリに割り当てるためにリンカー スクリプトが使用されます。これらについては、次のセクションで説明します。

MicroBlaze リンカー スクリプトで割り当てられるセクション

表 2-7 に、MicroBlaze リンカー スクリプトで割り当てられる入力セクションを示します。

表 2-7: リンカー スクリプトで割り当てられるセクション

セクション	説明
<code>.vectors.reset</code>	リセット ベクター コード
<code>.vectors.sw_exception</code>	ソフトウェア例外ベクター コード
<code>.vectors.interrupt</code>	ハードウェア割り込みベクター コード
<code>.vectors.hw_exception</code>	ハードウェア例外ベクター コード
<code>.text</code>	関数のコードおよびグローバル アセンブリ文からのプログラム命令
<code>.rodata</code>	読み出し専用変数
<code>.sdata2</code>	初期値を持つ読み出し専用の小さいスタティックおよびグローバル変数
<code>.data</code>	初期値を持つスタティックおよびグローバル変数。ブート コードによりゼロに初期化されます。
<code>.sdata</code>	初期値を持つ小さいスタティックおよびグローバル変数
<code>.sbss2</code>	初期値のない読み出し専用の小さいスタティックおよびグローバル変数。ブート コードによりゼロに初期化されます。
<code>.sbss</code>	初期値のない小さいスタティックおよびグローバル変数。ブート コードによりゼロに初期化されます。
<code>.bss</code>	初期値のないスタティックおよびグローバル変数。ブート コードによりゼロに初期化されます。
<code>.heap</code>	ヒープ用に定義されたメモリのセクション
<code>.stack</code>	スタック用に定義されたメモリのセクション

リンカー スクリプトを記述またはカスタマイズする際のヒント

カスタム リンカー スクリプトを記述する場合は、次の点に留意してください。

- ベクター セクションが MicroBlaze ハードウェアで定義された適切なメモリに割り当てられていることを確認します。
- スタックおよびヒープは `.bss` セクションに配置します。`_stack` 変数をこの領域の `_STACK_SIZE` の後に設定し、`_heap_start` 変数を `_STACK_SIZE` の後の次のロケーションに設定します。スタックおよびヒープは、ハードウェアおよびシミュレーションで初期化する必要がないので、`_bss_end` 変数は `.bss` および `COMMON` の後に定義します。

注記: `.bss` セクションの境界にはスタックおよびヒープは含まれません。

- `_SDATA_START__`、`_SDATA_END__`、`SDATA2_START`、`_SDATA2_END__`、`_SBSS2_START__`、`_SBSS2_END__`、`_bss_start`、`_bss_end`、`_sbss_start`、および `_sbss_end` 変数は、それぞれ `.sdata`、`.sdata2`、`.sbss2`、`.bss`、`.sbss` セクションの最初と最後に定義する必要があります。
- ANSI C では、初期化されないメモリはすべてスタートアップに初期化する必要があります (スタックおよびヒープでは不要)。提供されている標準の CRT では、1 つの `.bss` セクションが 0 に初期化されると想定されます。複数の `.bss` セクションがある場合は、この CRT は使用できません。その場合は、すべての `.bss` セクションを初期化する CRT を作成する必要があります。

スタートアップ ファイル

コンパイラで実行ファイルを生成する際、最後のリンク コマンドにはコンパイル済みのスタートアップ ファイルおよびエンド ファイルが含まれます。スタートアップ ファイルは、アプリケーション コードが実行される前に、言語およびプラットフォーム環境を設定します。スタートアップ ファイルでは、通常次の処理が実行されます。

- 必要に応じて、リセット、割り込み、および例外ベクターを設定します。
- スタック ポインター、スモール データ アンカー、およびその他のレジスタを設定します。詳細は、[34 ページの表 2-8](#)を参照してください。
- BSS メモリ領域を 0 にクリアします。
- C++ コンストラクタなどの言語初期化関数を呼び出します。
- ハードウェア サブシステムを初期化します。たとえば、プログラムのプロファイルが作成される場合は、プロファイル タイマーを初期化します。
- `main` プロシージャの引数を設定し、呼び出します。

エンド ファイルには、プログラムの終わりに実行する必要のあるコードが含まれています。エンド ファイルでは、通常次の処理が実行されます。

- C++ デストラクタなどの言語クリーンアップ関数を呼び出します。
- ハードウェア サブシステムの初期化を解除します。たとえば、プログラムのプロファイルが作成されている場合は、プロファイル サブシステムをクリーンアップします。

[表 2-8](#) には、C ランタイム ファイルのレジスタ名、値、およびその説明がまとめられています。

表 2-8: C ランタイム ファイルでのレジスタの初期化

レジスタ	値	説明
r1	<code>_stack-16</code>	スタック ポインター レジスタ。16 バイトの最初の負のオフセットでスタック領域の下部をポイントするよう初期化されます。この 16 バイトは、引数を渡すのに使用されます。
r2	<code>_SDA2_BASE</code>	読み出し専用のスモール データ領域アンカー アドレス。
r13	<code>_SDA_BASE_</code>	読み出し/書き込み可能なスモール データ領域アンカー アドレス。
その他のレジスタ	Undefined	その他のレジスタの値は定義されていません。

この後、さまざまなアプリケーション モードで使用する初期化ファイルについて説明します。この情報は、アプリケーションのスタートアップ コードを理解したい、または変更したいという上級者ユーザーを対象にしています。

MicroBlaze では、C ランタイム初期化に 2 つの段階があります。最初の段階では主にベクターが設定され、その後に第 2 段階が開始します。また、アプリケーション モードによっては、`exit` スタブも提供されます。

第 1 段階の初期化ファイル

crt0.o

ブートローダーまたはデバッグ スタブを使用せずに、スタンドアロンで実行されるプログラムに対して使用します。このファイルは、リセット、割り込み、例外、およびハードウェア例外ベクターを指定し、第 2 段階のスタートアップ ルーチン `_crtinit` を呼び出します。`_crtinit` から戻ると、`_exit` ラベルで無限ループを実行することによりプログラムを終了します。

crt1.o

アプリケーションをデバッグする際に使用します。ブレークポイントおよびリセット ベクター以外のすべてのベクターを指定し、第 2 段階のスタートアップ ルーチン `_crtinit` を呼び出します。

crt2.o

実行ファイルがブートローダーで読み込まれる場合に使用します。リセット ベクター以外のすべてのベクターを指定し、第 2 段階のスタートアップ ルーチン `_crtinit` を呼び出します。`_crtinit` から戻ると、`_exit` ラベルで無限ループを実行することによりプログラムを終了します。リセット ベクターは指定されないため、プロセッサのリセット時には、ブートローダーによりプログラムが再度読み込まれ、開始されます。

crt3.o

実行ファイルでベクターを使用せず、コード サイズを小さくする場合に使用します。リセット ベクターのみを指定し、第 2 段階のスタートアップ ルーチン `_crtinit` を呼び出します。`_crtinit` から戻ると、`_exit` ラベルで無限ループを実行することによりプログラムを終了します。ほかのベクターは指定されないため、リンクの段階で割り込み処理および例外処理に関連するルーチンが含まれることはなく、コード サイズが小さくなります。

第 2 段階の初期化ファイル

C 標準に従い、すべてのグローバルおよびスタティック変数は 0 に初期化する必要があります。これは上記すべての CRT に必要な共通の機能です。このため、別のルーチン `_crtinit` が呼び出されます。このルーチンは、プログラムの `.bss` セクションのメモリを初期化します。`_crtinit` ルーチンはラッパー ファイルでもあり、`main` プロシージャも呼び出します。`main` プロシージャを呼び出す前に、ほかの初期化関数が呼び出される場合もあります。`_crtinit` ルーチンは、次のスタートアップ ファイルにより提供されます。

crtinit.o

このデフォルトの第 2 段階の C スタートアップ ファイルは、次の処理を実行します。

1. `.bss` セクションを 0 にクリアします。
2. `_program_init` を呼び出します。
3. コンストラクタ関数 (`_init`) を呼び出します。
4. `main` プロシージャの引数を設定し、`main` を呼び出します。
5. デストラクタ関数 (`_fini`) を呼び出します。
6. `_program_clean` を呼び出し、戻ります。

pgcrtinit.o

これはプロファイルの作成時に使用され、次の処理を実行します。

1. .bss セクションを 0 にクリアします。
2. `_program_init` を呼び出します。
3. `_profile_init` を呼び出し、プロファイル ライブラリを初期化します。
4. コンストラクタ関数 (`_init`) を呼び出します。
5. `main` プロシージャの引数を設定し、`main` を呼び出します。
6. デストラクタ関数 (`_fini`) を呼び出します。
7. `_profile_clean` を呼び出し、プロファイル ライブラリをクリーンアップします。
8. `_program_clean` を呼び出し、戻ります。

sim-crtinit.o

コンパイラで `-mno-clearbss` オプションが設定されている場合に使用され、次の処理を実行します。

1. `_program_init` を呼び出します。
2. コンストラクタ関数 (`_init`) を呼び出します。
3. `main` プロシージャの引数を設定し、`main` を呼び出します。
4. デストラクタ関数 (`_fini`) を呼び出します。
5. `_program_clean` を呼び出し、戻ります。

sim-pgcrtinit.o

プロファイルの作成時に、コンパイラで `-mno-clearbss` オプションが設定されている場合に使用され、次の処理を実行します。

1. `_program_init` を呼び出します。
2. `_profile_init` を呼び出し、プロファイル ライブラリを初期化します。
3. コンストラクタ関数 (`_init`) を呼び出します。
4. `main` プロシージャの引数を設定し、呼び出します。
5. デストラクタ関数 (`_fini`) を呼び出します。
6. `_profile_clean` を呼び出し、プロファイル ライブラリをクリーンアップします。
7. `_program_clean` を呼び出し、戻ります。

その他のファイル

コンパイラは、C++ 言語をサポートするため、特定の標準スタート ファイルおよびエンド ファイルも使用します。`crti.o`、`crtbegin.o`、`crtend.o`、および `crttn.o` がそれに当たります。これらの標準コンパイラ ファイルは、`.init`、`.fini`、`.ctors`、および `.dtors` セクションの内容を指定します。

スタートアップ ファイルの変更

初期化ファイルは、コンパイル済みのファイルおよびソース ファイルの両方で提供されています。コンパイル済みのオブジェクト ファイルは、コンパイラ ライブラリ ディレクトリに含まれます。MicroBlaze の GNU コンパイラの初期化ファイルのソースは、<XILINX_>/SDK/<version>/data/embeddedsw/lib/microblaze/src/ ディレクトリにあります。ここで、<XILINX_> は Vivado のインストールパス、<version> は SDK のリリースバージョンです。

カスタム スタートアップ ファイルを使用するには、ソース ディレクトリにあるファイルを、アプリケーション ソースの一部として含める必要があります。また、ファイルを .o ファイルに統合して、共有エリアに配置することも可能です。標準ファイルではなく新しく作成したオブジェクト ファイルを参照する場合は、mb-gcc の実行時に -B directory -name を使用します。

デフォルトのスタートアップ ファイルが使用されないようにするには、コンパイルの最後の行に -nostartfiles を追加します。

注記: crt1.o、crtbegin.o などのコンパイラ標準 CRT ファイルは、ソース コードが提供されないもので、これらのファイルは、インストール ディレクトリに含まれているものをそのまま使用してください。これらのファイルは、最後のリンク コマンドに含める必要がある場合があります。

C プログラムのスタートアップ コード サイズの縮小

C プログラムのコード サイズの制限が厳しい場合は、オーバーヘッドの原因となるあらゆるものを取り除く必要があります。このセクションでは、C プログラムで不要な C++ コンストラクタまたはデストラクタ コードによるオーバーヘッドを削減する方法を説明します。次の変更を加えることにより、コードのサイズを約 220 バイト縮小できます。

1. 前のセクションで説明したように、インストール領域からスタートアップ ファイルのカスタム コピーを作成します。アプリケーションに適した crtn.s および xcrtinit.s をコピーします。たとえば、アプリケーションがブートローダーを使用して読み込まれ、プロファイルが作成される場合は、インストール領域から crt2.s および pg-crtinit.s をコピーします。

2. pg-crtinit.s から次の行を削除します。

```
brlid r15, __init
/* Invoke language initialization functions */
nop
```

および

```
brlid r15, __fini
/* Invoke language cleanup functions */
nop
```

これにより、コンストラクタおよびデストラクタの処理に使用されるコードは参照されなくなり、コード サイズが小さくなります。

3. これらのファイルを .o ファイルにコンパイルして任意のディレクトリに配置するか、アプリケーション ソースの一部として含めます。
4. コンパイラに -nostartfiles オプションを追加します。特定のフォルダーにファイルを統合する場合は、-B directory オプションも使用します。
5. アプリケーションをコンパイルします。

アプリケーションを異なるモードで実行する場合は、34 ページの「スタートアップ ファイル」の説明に従って、適切な CRT ファイルを選択する必要があります。

コンパイラ ライブラリ

mb-gcc コンパイラには、GNU C 標準ライブラリと GNU 数学ライブラリが必要です。Vivado には、あらかじめコンパイルされたこれらのライブラリが含まれています。MicroBlaze のハードウェア コンフィギュレーションに基づき、MicroBlaze の CPU ドライバーの該当バージョンがコピーされます。使用するライブラリのバージョンを選択するには、次のフォルダーを確認します。

```
$XILINX_/gnu/microblaze/<platform>/microblaze-xilinx-elf/lib
```

ファイル名は、コンパイラのオプションとライブラリのコンパイルに使用されたコンフィギュレーションに基づいて付けられています。たとえば、libc_m_bs.a は、ハードウェア乗算器とバレルシフターをイネーブルにしてコンパイルされた C ライブラリです。

表 2-9 に、使用されているエンコードとそのエンコードで指定されるライブラリのコンフィギュレーションを示します。

表 2-9: コンパイラ フラグ上のエンコードされたライブラリ ファイル名

エンコード	説明
_bs	バレルシフター用にコンフィギュレーション
_m	ハードウェア乗算器用にコンフィギュレーション
_p	パターンコンパレータ用にコンフィギュレーション

注意が必要なのは、数学ライブラリ ファイル (libm*.a) です。C 標準では、倍精度の浮動小数点演算を使用するために、sin() および cos() などの共通の数学ライブラリ関数が必要です。ただし、倍精度の浮動小数点演算では、MicroBlaze 用にオプションで提供されている単精度の浮動小数点機能を完全に利用できない可能性があります。

Newlib 数学ライブラリには、単精度演算を使用してこれらの数学関数をインプリメントするバージョンがあります。これらの単精度ライブラリでは MicroBlaze ハードウェア浮動小数点ユニット (FPU) を直接使用できるので、パフォーマンスが向上する可能性があります。

アプリケーションで標準精度が必要ではなく、パフォーマンスを向上させたい場合は、リンクされたライブラリのバージョンを手動で変更できます。

デフォルトでは、CPU ドライバーにより倍精度バージョンのライブラリ (libm*_fpu.a) が IP インテグレーター プロジェクトにコピーされます。

単精度バージョンを利用するには、対応する libm*_fps.a をコピーするカスタム CPU ドライバーを作成します。この場合、対応する libm*_fps.a ファイルを libm.a としてプロセッサのライブラリ フォルダー (microblaze_0/lib など) にコピーします。

使用するライブラリをコピーしたら、アプリケーション ソフトウェア プロジェクトを再ビルドします。

スレッド セーフ

Vivado に含まれる MicroBlaze の C ライブラリおよび数学ライブラリは、マルチスレッド環境で使用するようにはビルドされていません。printf()、scanf()、malloc()、free() などの共通 C ライブラリ関数はスレッドセーフではなく、システムで回復不可能なランタイム エラーを引き起こすことがあります。マルチスレッド環境で Vivado ライブラリを使用する場合は、相互排他的なメカニズムを使用してください。

コマンド ライン引数

MicroBlaze プロセッサ プログラムでは、コマンド ライン引数を使用できません。コマンド ライン引数 `argc` および `argv` は、C ランタイム ルーチンで 0 に初期化されます。

割り込みハンドラー

割り込みハンドラー、通常のサブルーチン呼び出しとは別の方法でコンパイルされます。割り込みハンドラーでは、不揮発性レジスタだけでなく、使用されている揮発性レジスタも保存する必要があります。また、割り込みが発生した際に、マシン ステータス レジスタ (RMSR) の値も保存する必要があります。

interrupt_handler 属性

サブルーチンと割り込みハンドラを区別するため、`mb-gcc` はコードの宣言部に `interrupt_handler` 属性があるかどうかをチェックします。この属性は、次のように定義されています。

```
void function_name () __attribute__((interrupt_handler));
```

注記: 割り込みハンドラの属性は、プロトタイプ内でのみ指定し、定義には含めません。

割り込みハンドラーは、揮発性レジスタを使用するほかの関数を呼び出すこともあります。揮発性レジスタで正しい値を保持するため、ハンドラーが非リーフ関数である場合は、すべての揮発性レジスタが保存されます。

注記: 非リーフ関数とは、ほかのサブルーチンを呼び出す関数のことです。

割り込みハンドラーは、MSS (Microprocessor Software Specification) ファイルで定義されます。これらの定義により、割り込みハンドラ関数に属性が自動的に追加されます。

割り込みハンドラーは `rtid` 命令を使用して、割り込みで中断された関数に戻ります。

save_volatiles 属性

`save_volatiles` 属性は、`interrupt_handler` 属性と似ていますが、割り込み処理から戻るのに `rtid` ではなく `rtsd` を使用します。

この属性を使用すると、非リーフ関数の場合はすべての揮発性レジスタが保存され、リーフ関数の場合は使用されている揮発性レジスタのみが保存されます。

```
void function_name () __attribute__((save_volatiles));
```

fast_interrupt

MicroBlaze コンパイラには、`interrupt_handler` 属性に類似した `fast_interrupt` という属性があります。高速割り込みがあると、MicroBlaze は、固定アドレス `0x10` に飛ぶのではなく、割り込みルーチン アドレスに飛びます。

標準割り込みとは異なり、`fast_interrupt` 属性が C 関数で 사용되는場合は、MicroBlaze は最小限のレジスタしか保存しません。

```
void function_name () __attribute__((fast_interrupt));
```

表 2-10: 割り込みハンドラーの使用法

属性	関数
interrupt_handler	マシン ステータス レジスタ、不揮発性レジスタ、およびすべての揮発性レジスタを保存します。割り込みから戻るには rtid を使用します。割り込みハンドラ関数がリーフ関数の場合は、関数で使用された揮発性レジスタのみが保存されます。
save_volatiles	interrupt_handler と似ていますが、割り込みから戻るのに rtid ではなく rtsd が使用されます。
fast_interrupt	interrupt_handler と似ていますが、固定アドレス 0x10 に飛ぶのではなく、割り込みルーチン アドレスに直接飛びます。

Arm Cortex-A9 コンパイラの使用法とオプション

Sourcery CodeBench Lite for Xilinx EABI を使用し、Arm プロセッサをコンパイルすることができます。

Sourcery CodeBench には、次のコンポーネントをすべて含む、GNU ツールチェーンが含まれています。

- CodeSourcery の共通スタートアップ コード シーケンス
- CodeSourcery Debug Sprite for Arm
- GNU バイナリ ユーティリティ (Binutils)
- GNU C コンパイラ (GCC)
- GNU C++ コンパイラ (G++)
- GNU C++ ランタイム ライブラリ (Libstdc++)
- GNU デバッガー (GDB)
- Newlib C ライブラリ

使用法

コンパイル

```
arm-xilinx-eabi-gcc -c file1.c -I<include_path> -o file1.o
arm-xilinx-eabi-gcc -c file2.c -I<include_path> -o file2.o
```

リンク

```
arm-xilinx-eabi-gcc -Wl,-T -Wl,lscrip.ld -L<libxil.a path> -o "App.elf"file1.o
file2.o -Wl,--start-group,-lxil,-lgcc,-lc,--end-group
```

上記のコマンドで使用されているフラグの説明については、次のコマンドのいずれかを使用して、コンパイラのヘルプを参照してください。

- arm-xilinx-eabi-gcc --help
- arm-xilinx-eabi-gcc -v --help
- arm-xilinx-eabi-gcc --target-help

コンパイラ オプション

Arm 関連のフラグを使用して適用できる、GNU コンパイラ オプションについては、GNU ウェブ サイト <http://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html> を参照してください。これらのフラグは、要件に基づき上記の手順で使用できます。

Arm の GCC コンパイラ オプションはすべて、上記のリンク先にリストされています。しかし、実際のサポートは使用しているプロセッサ (この場合は Arm Cortex A9) およびコンパイラ ツールチェーンによって異なります。

次は、その例です。

Sourcery CodeBench Lite for Xilinx EABI は、`-mhard-float` (`-mfloat-abi=hard`) をサポートしません。soft および `softfp` の浮動小数点オプションのみがサポートされています。

ツールチェーンの詳細については、次の SDK インストール パスにある資料を参照してください。

```
<Xilinx_Vivado_Installation_Path>\SDK\<2016.1>\gnu\arm\nt\share\doc
```

その他の注意事項

C++ コードのサイズ

最新のオープン ソース C++ 標準ライブラリ (libstdc++-v3) を含む GCC ツールでは、同等の C プログラムに比べて生成されるコードおよびデータ片が大きくなる場合があります。このオーバーヘッドの大部分は、例外処理およびランタイム型情報のコードおよびデータによるものです。C++ アプリケーションによっては、これらの機能は必要ありません。

このオーバーヘッドをなくし、コード サイズを最適化するには、`-fno-exceptions` および `-fno-rtti` オプションを使用します。これらのオプションは、アプリケーション要件やこれらの言語の機能に精通している場合のみに使用することをお勧めします。使用可能なコンパイラ オプションおよびそれらのオプションによる影響については、GCC のマニュアルを参照してください。

C++ プログラムには、より複雑な言語機能およびライブラリ ルーチンのため、ダイナミック メモリの要件 (スタックおよびヒープ サイズ) が厳しくなっている場合があります。

多くの C++ ライブラリ ルーチンは、ヒープからのメモリの割り当てを要求します。C++ プログラムに必要なヒープサイズおよびスタック サイズが満たされていることを確認してください。

C++ 標準ライブラリ

C++ 標準ライブラリは、C++ 標準により定義されます。これらのプラットフォーム機能には、デフォルトのザイリンクス Vivado ソフトウェア プラットフォームでは使用できないものもあります。たとえば、ファイル I/O は明確に定義された STDIN/STDOUT ストリームでしかサポートされません。また、ロケール関数、スレッド セーフなどの機能もサポートされません。

注記: C++ 標準ライブラリは、マルチスレッド環境で使用するようには構築されていません。new、delete などの共通 C++ 関数はスレッド セーフではありません。マルチスレッド環境で C++ 標準ライブラリを使用する場合は、注意が必要です。

GNU C++ 標準ライブラリの詳細は、GNU の ウェブ サイトのマニュアルを参照してください。

位置独立コード (PIC)

MicroBlaze コンパイラでは、再配置可能な位置独立コードを生成する `-fPIC` オプションがサポートされています。

これらの機能はザイリンクスのコンパイラでサポートされていますが、Vivado ではスタンドアロン プラットフォームしか提供されないため、ほかのライブラリおよびツールではサポートされません。位置独立コードは、ローダーまたはデバッガーでは認識されず、ランタイムでの再配置は実行されません。これらのコードの機能は、ザイリンクス ライブラリ、スタートアップ ファイル、およびその他のツールでサポートされません。サードパーティ OS ベンダーのツールでは、これらの機能を標準で使用できるものがあります。

その他のオプションおよび機能

`-fprofile-arcs` など、その他のオプションおよび機能は、ザイリンクスの Vivado コンパイラおよびプラットフォームでサポートされていない可能性があります。一部の機能は、オープン ソース GCC で定義されているように試験段階であり、不適切に使用すると、不正なコードが生成されることがあります。特定の機能の詳細は、GCC のマニュアルを参照してください。

Xilinx System Debugger

Xilinx® System Debugger を使用すると、実行中にプログラムに何が起きているのかを確認できるようになります。プロセッサを停止するブレークポイントまたはウォッチポイントの設定、プログラムのステップ実行、プログラム変数およびスタックの確認、システム内のメモリ内容の確認などを実行できます。

Xilinx System Debugger では、SDK およびコマンドライン インターフェイス (CLI) を使用したデバッグがサポートされます。

SDK System Debugger

SDK System Debugger では、ザイリンクスの `hw_server` が基本的なデバッグ エンジンとして使用されます。SDK では各ユーザー インターフェイス アクションが一連の TCF コマンドに変換され、システム デバッガーからの出力が処理され、デバッグされている現在のプログラムの状態が表示されます。SDK ではザイリンクス `hw_server` を使用してハードウェア上のプロセッサと通信します。次の図はこのデバッグ ワークフローを示しています。

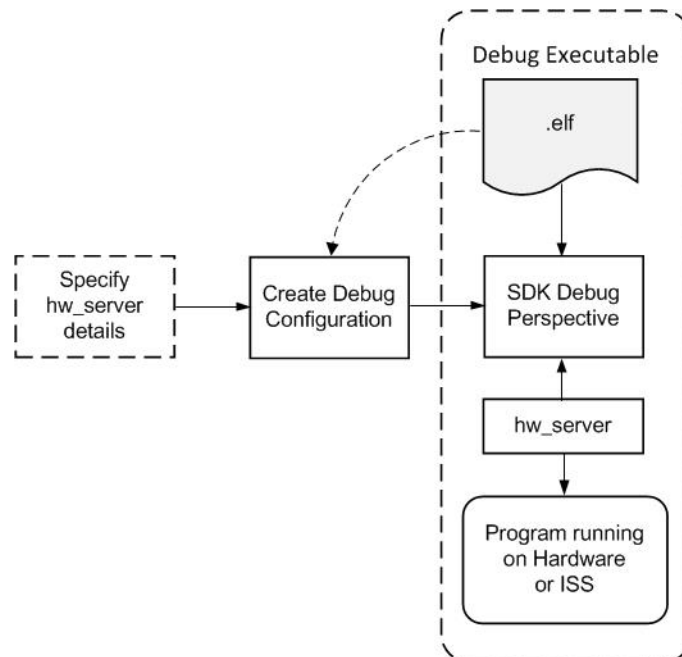


図 3-1: デバッグ ワークフロー

ワークフローは、次のコンポーネントから構成されます。

- **実行可能な ELF ファイル:** ユーザー アプリケーションをデバッグするには、デバッグ用にコンパイルされた ELF (Executable and Linkable Format) ファイルを使用する必要があります。デバッグ ELF ファイルには、デバッガーへの追加のデバッグ情報が含まれ、ソース コードとその元のソースから生成された 2 進数間が直接関連付けられます。
- **デバッグ コンフィギュレーション:** デバッグ セッションを開始するには、SDK でデバッグ コンフィギュレーションを作成する必要があります。このコンフィギュレーションには、実行ファイル名、デバッグするプロセス ターゲット、およびその他の情報など、デバッグ セッションを開始するに必要なオプションが含まれます。
- **SDK の [Debug] パースペクティブ:** [Debug] パースペクティブを使用すると、ワークベンチでプログラムのデバッグまたは実行が管理できます。プログラムの実行は、ブレークポイントを設定し、起動したプログラムを一時停止し、コードをステップ スルーし、変数の内容を検証することで制御できます。

コードの修正、実行ファイルの構築、プログラムのデバッグといったサイクルは、SDK で繰り返すことができます。

注記: コンパイル後にソースを編集すると、デバッグ情報がそのソースに直接関連付けられているので、その行番号はステップ実行されなくなります。同様に、最適化された 2 進数をデバッグしても、実行トレースで予想外にジャンプしてしまうことがあります。

SDK System Debugger の詳細は、SDK ヘルプ [\[参照 1\]](#) を参照してください。

Xilinx System Debugger コマンドライン インターフェイス (XSDB)

Xilinx System Debugger コマンドライン インターフェイス (XSDB) は、ザイリンクス `hw_server` およびその他のザイリンクスで使用される TCF サーバーに、使用しやすいインタラクティブなスクリプト記述可能なコマンドライン インターフェイスを提供します。XSDB は TCF サーバーと対話するので、TCF サーバーでサポートされる機能を最大限に利用できます。

XSDB は、次を実行します。

- システム全体との対話が可能
- ハード化されたプログラマブル ロジックのソフトウェア エンジニアの観点をサポート
- パフォーマンス計測機能を提供
- `hw_server` とその他の TCF サーバーを統合

SDK System Debugger の詳細は、SDK ヘルプ [\[参照 1\]](#) を参照してください。

フラッシュ メモリのプログラム

概要

プログラム フラッシュ ユーティリティは、ボードのフラッシュ メモリを消去およびプログラムするために使用できます。また、ブランク チェックおよび検証など、削除およびプログラム機能を検証するのに便利なその他のオプションもあります。ブランク チェックをオンにすると、フラッシュからの内容を読み出して、フラッシュ パーツが空かどうかチェックされます。同様に、検証機能を使用すると、データを読み戻して、プログラムされたデータと比較して、そのデータが正しく書き込まれたかどうかを確認されます。

Zynq デバイス

プログラム フラッシュ ユーティリティでは、QSPI、NAND、および NOR などのタイプのフラッシュのプログラムがサポートされます。QSPI は、QSPI Single、QSPI Dual Parallel、および QSPI Dual Stacked などのさまざまなコンフィギュレーションで使用できます。NAND および NOR タイプの場合は、FSBL ファイルを指定する必要があります。

Bootgen からのブート イメージをプログラムできます。Bootgen では、FSBL (First Stage Boot Loader)、ビットストリーム (Zynq® の PL 部分をコンフィギュレーションするため)、アプリケーション、RTOS、およびその他のデータ ファイルなどのコンポーネントがまとめられます。

Zynq の場合、プロセッサがリセットから抜け出すと、BootROM で制御されるので、フラッシュからの FSBL がオンチップ メモリにコピーされて、制御が渡されます。FSBL が実行を開始すると、フラッシュからのビットストリームがコピーされ PL がコンフィギュレーションされます。PL がコンフィギュレーションされたら、FSBL で次のパーティション (たとえばアプリケーション) がフラッシュから DDR へコピーされ、制御がアプリケーションに渡され、アプリケーションの実行が開始されます。Linux を読み込むには、もう 1 つのパーティションとして U-Boot を使用できます。

その他のデバイス

フラッシュは、Parallel Flash (BPI) と Serial Flash (SPI) に広く分類されます。SPI フラッシュも BPI フラッシュも Micron、Spansion などのさまざまなメーカーから取得できます。フラッシュでは、次をプログラムできます。

- アプリケーションの実行イメージ
- FPGA のハードウェア ビットストリーム
- ファイル システム イメージ、サンプル データやアルゴリズム テーブルなどのデータ ファイル

アプリケーションの実行イメージが最も一般的です。デザインのプロセッサのリセットが完了すると、ブロック RAM のプロセッサのリセット位置に保存されている実行コードが開始します。通常ブロック RAM のサイズは数 KB でソフトウェア アプリケーションのイメージ全体を保存するには小さすぎるので、フラッシュ メモリ (単位は MB) を使用します。小型のブートローダーはブロック RAM にフィットするよう設計されています。プロセッサはリセット時にこのブートローダーを実行し、ソフトウェア アプリケーション イメージがフラッシュから外部メモリへとコピーされます。この後、制御はブートローダーからソフトウェア アプリケーションへと移り、実行が継続されます。

プロジェクトで作成したソフトウェア アプリケーションのフォーマットは、ELF (Executable Linked Format) です。フラッシュからソフトウェア アプリケーションをブートロードする場合、ELF イメージを、SREC (モトローラ S レコード) などのブートロード可能なイメージフォーマットに変換する必要があります。こうすると、ブートローダーをシンプルに、またサイズも小さく抑えることができます。

プログラム フラッシュ ユーティリティ

プログラム フラッシュは、ソフトウェアおよびデータを使用してオンボードのシリアルおよびパラレル フラッシュ デバイスを削除してプログラムできるようにするコマンド ライン ユーティリティです。

使用法

```
program_flash <flash options> <cable device options>
```

フラッシュ オプション

オプション	説明
-f <image file>	フラッシュ メモリに書き込まれるイメージ (bin/mcs のみ)
-offset <address>	イメージの書き込まれるフラッシュ メモリ内のオフセット
-no_erase	プログラム前にフラッシュ メモリの消去なし
-erase_only	イメージ ファイルのサイズに応じてフラッシュを消去
-blank_check	フラッシュ メモリが消去されたかどうかを確認
-verify	フラッシュ メモリが正しくプログラムされたかどうかを確認
-fsbl <fsbl file>	NAND および NOR フラッシュ タイプの場合のみ (Zynq のみ)
-erase_sector <size>	消去セクターフラッシュが 64KB (バイト) 以外のフラッシュの場合

オプション	説明
<code>-flash_type <type></code>	<p>サポートされるフラッシュ メモリ タイプ:</p> <ul style="list-style-type: none"> • Zynq デバイス <ul style="list-style-type: none"> ◦ qspi_single ◦ qspi_dual_parallel ◦ qspi_dual_stacked ◦ nand_8 ◦ nand_16 ◦ nor • その他のデバイス <p>すべてのフラッシュ タイプをリストするには、<code>-partlist</code> コマンド ライン オプションを使用します。</p>
<code>-partlist <bpi spi> <micron spansion></code>	<p>その他 (Zynq 以外) のデバイスのフラッシュ パーツをすべてリスト</p> <ul style="list-style-type: none"> • <code>program_flash -partlist</code> - すべてのフラッシュをリスト • <code>program_flash -partlist bpi micron</code> - すべての Micron BPI フラッシュをリスト • <code>program_flash -partlist spi spansion</code> - Spansion SPI フラッシュをリスト

ケーブルおよびデバイス オプション

オプション	説明
<code>-cable type <type of cable></code> <code>esn <cable esn></code> <code>url <URL></code>	<ul style="list-style-type: none"> • <code>type <type of cable></code> - ケーブル タイプ (<code>xilinx_tcf</code>) を指定。 • <code>esn <cable esn></code> - ホスト マシンに接続された USB ケーブルの Electronic Serial Number (ESN) を指定。このオプションを使用すると、複数のケーブルがホスト マシンに接続されている場合に、USB ケーブルを識別できるようになります。 • <code>url <URL></code> - <code>hw_server/TCF</code> エージェントの URL 記述
<code>-debugdevice deviceNr <device position in jtag chain></code>	<p><code>-deviceNr</code> - デバイスの JTAG チェーン的位置。デバイス位置の番号は、1 から開始します。</p>

その他の注意事項

サポートされるフラッシュ パーツ (Zynq デバイス以外)

次の表は、Zynq デバイス以外でサポートされるフラッシュ パーツをすべてリストしています。パーツ名情報は、`-flash_type` コマンド ライン オプションを使用して渡されます。リストには、Micron 社および Spansion 社からの BPIx8、BPIx16、および SPI タイプのフラッシュが含まれます。フラッシュは、`-partlist` コマンド ライン オプションを使用すると、タイプ別 (BPI/SPI) または製造業者別 (Spansion/Micron) でフィルタリングすることができます。

表 4-1: サポートされるフラッシュ パーツ (Zynq デバイス以外)

S.No.	製造業者	パーツ名 (-flash_type)
1:	Spansion	s29gl128p-bpi-x16
2:	Spansion	s29gl256p-bpi-x16
3:	Spansion	s29gl512p-bpi-x16
4:	Spansion	s29gl01gp-bpi-x16
5:	Spansion	s29gl128s-bpi-x16
6:	Spansion	s29gl256s-bpi-x16
7:	Spansion	s29gl512s-bpi-x16
8:	Spansion	s29gl01gs-bpi-x16
9:	Spansion	s29gl128p-bpi-x8
10:	Spansion	s29gl256p-bpi-x8
11:	Spansion	s29gl512p-bpi-x8
12:	Spansion	s29gl01gp-bpi-x8
13:	Micron	28f640p30t-bpi-x16
14:	Micron	28f640p30b-bpi-x16
15:	Micron	28f128p30t-bpi-x16
16:	Micron	28f128p30b-bpi-x16
17:	Micron	28f256p30t-bpi-x16
18:	Micron	28f256p30b-bpi-x16
19:	Micron	28f512p30t-bpi-x16
20:	Micron	28f512p30e-bpi-x16
21:	Micron	28f512p30b-bpi-x16
22:	Micron	28f00ap30t-bpi-x16
23:	Micron	28f00ap30e-bpi-x16
24:	Micron	28f00ap30b-bpi-x16
25:	Micron	28f00bp30e-bpi-x16
26:	Micron	28f640p33t-bpi-x16
27:	Micron	28f640p33b-bpi-x16
28:	Micron	28f128p33t-bpi-x16

表 4-1: サポートされるフラッシュ パーツ (Zynq デバイス以外) (続き)

S.No.	製造業者	パーツ名 (-flash_type)
29:	Micron	28f128p33b-bpi-x16
30:	Micron	28f256p33t-bpi-x16
31:	Micron	28f256p33b-bpi-x16
32:	Micron	28f512p33t-bpi-x16
33:	Micron	28f512p33e-bpi-x16
34:	Micron	28f512p33b-bpi-x16
35:	Micron	28f00ap33t-bpi-x16
36:	Micron	28f00ap33e-bpi-x16
37:	Micron	28f00ap33b-bpi-x16
38:	Micron	28f128g18f-bpi-x16
39:	Micron	mt28gu256aax1e-bpi-x16
40:	Micron	mt28gu512aax1e-bpi-x16
41:	Micron	mt28gu01gaax1e-bpi-x16
42:	Micron	28f064m29ewh-bpi-x16
43:	Micron	28f064m29ewl-bpi-x16
44:	Micron	28f064m29ewt-bpi-x16
45:	Micron	28f064m29ewb-bpi-x16
46:	Micron	28f128m29ew-bpi-x16
47:	Micron	28f256m29ew-bpi-x16
48:	Micron	28f512m29ew-bpi-x16
49:	Micron	28f00am29ew-bpi-x16
50:	Micron	28f00bm29ew-bpi-x16
51:	Micron	28f064m29ewh-bpi-x8
52:	Micron	28f064m29ewl-bpi-x8
53:	Micron	28f064m29ewt-bpi-x8
54:	Micron	28f064m29ewb-bpi-x8
55:	Micron	28f128m29ew-bpi-x8
56:	Micron	28f256m29ew-bpi-x8
57:	Micron	28f512m29ew-bpi-x8
58:	Micron	28f00am29ew-bpi-x8
59:	Micron	28f00bm29ew-bpi-x8
60:	Spansion	s70gl02gp-bpi-x16
61:	Spansion	s70gl02gs-bpi-x16
62:	Spansion	s25fl032p-spi-x1_x2_x4
63:	Spansion	s25fl064p-spi-x1_x2_x4
64:	Spansion	s25fl132k-spi-x1_x2_x4

表 4-1: サポートされるフラッシュ パーツ (Zynq デバイス以外) (続き)

S.No.	製造業者	パーツ名 (-flash_type)
65:	Spansion	s25fl164k-spi-x1_x2_x4
66:	Spansion	s25fl128sxxxxxx0-spi-x1_x2_x4
67:	Spansion	s25fl128sxxxxxx1-spi-x1_x2_x4
68:	Spansion	s25fl256sxxxxxx0-spi-x1_x2_x4
69:	Spansion	s25fl256sxxxxxx1-spi-x1_x2_x4
70:	Spansion	s25fl512s-spi-x1_x2_x4
71:	Micron	mt25qu512-spi-x1_x2_x4
72:	Micron	mt25qu512-spi-x1_x2_x4_x8
73:	Micron	mt25ql512-spi-x1_x2_x4
74:	Micron	mt25ql512-spi-x1_x2_x4_x8
75:	Micron	mt25ql01g-spi-x1_x2_x4
76:	Micron	mt25ql01g-spi-x1_x2_x4_x8
77:	Micron	mt25ql02g-spi-x1_x2_x4
78:	Micron	mt25ql02g-spi-x1_x2_x4_x8
79:	Micron	mt25qu01g-spi-x1_x2_x4
80:	Micron	mt25qu01g-spi-x1_x2_x4_x8
81:	Micron	mt25qu02g-spi-x1_x2_x4
82:	Micron	mt25qu02g-spi-x1_x2_x4_x8
83:	Micron	n25q128-3.3v-spi-x1_x2_x4
84:	Micron	n25q128-1.8v-spi-x1_x2_x4
85:	Micron	n25q256-3.3v-spi-x1_x2_x4
86:	Micron	n25q256-1.8v-spi-x1_x2_x4_x8
87:	Micron	n25q256-1.8v-spi-x1_x2_x4
88:	Micron	n25q32-3.3v-spi-x1_x2_x4
89:	Micron	n25q32-1.8v-spi-x1_x2_x4
90:	Micron	n25q64-3.3v-spi-x1_x2_x4
91:	Micron	n25q64-1.8v-spi-x1_x2_x4

ブートローダー アプリケーション用に ELF ファイルを SREC に変換する方法

mb-objcopy ユーティリティを使用すると、ELF ファイルから SREC フォーマット ファイルを作成できます。SREC フォーマット アプリケーションは、特定のオフセットでフラッシュに格納できます。SREC ブートローダーでは、これらのアプリケーションを読み込んで、ロードして、実行できます。たとえば、myexecutable.elf ファイルを含むフォルダーに移動して次を実行します。

```
mb-objcopy -O srec myexecutable.elf myexecutable.srec
```

これにより、SREC ファイルが生成されます。mb-objcopy は SDK に含まれる GNU バイナリ ユーティリティです。

プログラム用に SREC/ELF/BIT ファイルを BIN/MCS ファイルへ変換する方法

ザイリンクス Bootgen ユーティリティを使用すると、さまざまなファイルから BIN/MCS ファイルを作成できます。

```
bootgen -arch fpga -image <input.bif> -o <output.bin/mcs> -interface <options>
```

Bootgen オプション

オプション	説明
-image <input.bif>	入力ファイルに関する情報を含むブート イメージ フォーマット ファイルを入力
-o <output.bin/mcs>	出力ファイル パスおよびフォーマット <ul style="list-style-type: none"> • -o output.bin - output という名前の BIN ファイル • -o output.mcs - output という名前の MCS ファイル
-interface <options>	フラッシュからプログラムしてブートするインターフェイス <ul style="list-style-type: none"> • spi • bpix8 • bpix16 • smapx8 • smapx16 • smapx32

例

1. ELF ファイルを BIN ファイルに変換します。

```
bootgen -arch fpga -image elf_bin_all.bif -o boot.bin -interface spi
```

elf_bin_all.bif ファイルの内容は次のとおりです。

```
image:
{
hello.elf
}
```

2. SREC ファイルを BIN ファイルに変換します。

```
bootgen -arch fpga -image srec_bin_all.bif -o boot.bin -interface spi
```

srec_bin_all.bif ファイルの内容は次のとおりです。

```
image:
{
hello.elf.srec
}
```

3. BIT ファイルを BIN ファイルに変換します。

```
bootgen -arch fpga -image bit_bin_all.bif -o boot.bin -interface spi
```

bit_bin_all.bif ファイルの内容は次のとおりです。

```
image:
{
system.bit
}
```

Zynq デバイスのイメージの作成

ザイリンクス Bootgen は、Zynq デバイスのイメージを作成するために使用します。さまざまなコンポーネントがまとめられて、ブート イメージが作成されます。コンポーネントはオプションで暗号化したり、認証したり、チェックサム計算したりできます。ブート イメージを作成するには、さまざまなオプションがあります。

詳細は、『Zynq-7000 SoC ソフトウェア開発者向けガイド』(UG821) [\[参照 8\]](#) を参照してください。

GNU ユーティリティ

この付録では、Vivado® Design Suite で使用可能な GNU ユーティリティについて説明します。

MicroBlaze プロセッサ用の汎用ユーティリティ

cpp

C および C++ コードのプリプロセッサ。プリプロセッサは GCC (GNU コンパイラ) により自動的に実行され、file-include、define などの指示子をインプリメントします。

gcov

GCC と共に使用し、ユーザー プログラムのテスト範囲のプロファイル作成および解析を実行します。gprof プロファイル作成プログラムでも使用できます。

注記: gcov は IP インテグレーターまたは SDK ではサポートされていませんが、カスタム テスト フローを実行する必要があるケース用に提供されています。

MicroBlaze プロセッサ用ユーティリティ

MicroBlaze™ 用ユーティリティには接頭辞 **mb** が付いており、次のものがあります。

mb-addr2line

実行ファイルのデバッグ情報を使用して、プログラム アドレスを対応する行番号とファイル名に変換するプログラム。

mb-ar

アーカイブからファイルを作成、変更、および抽出するプログラム。アーカイブは、通常ライブラリのオブジェクト ファイルなどの複数のファイルを含むファイルです。

mb-as

アセンブラー プログラム。

mb-c++

mb-gcc と同じクロス コンパイラで、プログラム言語が C++ に設定されている場合に実行されます。**mb-g++** と同じです。

mb-c++filt

アセンブリ リストの C++ および Java 関数名をデマングルするプログラム。

mb-g++

mb-gcc と同じクロス コンパイラで、プログラム言語が C++ に設定されている場合に実行されます。**mb-c++** と同じです。

mb-gasp

アセンブラー プログラムのマクロ プリプロセッサ。

mb-gcc

C および C++ プログラムのクロス コンパイラ。ファイル拡張子から、使用されているプログラム言語を自動的に認識します。

mb-gdb

プログラムのデバッガー。

mb-gprof

プログラムの各部分にどれだけの時間がかかるかを解析するプロファイル生成プログラム。ランタイムを最適化するのに有益です。

mb-ld

リンカー プログラム。ライブラリ ファイルとオブジェクト ファイルを結合し、必要なリロケーションを実行して、実行ファイルを生成します。

mb-nm

オブジェクト ファイルのシンボルをリストするプログラム。

mb-objcopy

オブジェクト ファイルの内容をあるフォーマットから別のフォーマットに変換するプログラム。

mb-objdump

オブジェクト ファイルの情報を表示するプログラム。プログラムのデバッグにおいて有益で、正しいコードおよびデータが正しいメモリ ロケーションにあるかどうかを検証するのに使用されます。

mb-ranlib

アーカイブ ファイルのインデックスを作成し、アーカイブ ファイルに追加するプログラム。アーカイブで示されるライブラリへのリンク プロセスを高速化できます。

mb-readelf

ELF (Executable Linked Format) ファイルの情報を表示するプログラム。

mb-size

オブジェクト ファイルの各セクションのサイズをリストするプログラム。コードおよびデータのスタティック メモリ要件を判断するのに便利です。

mb-strings

バイナリ ファイルの内容を判断するのに便利なプログラム。オブジェクト ファイルに含まれる表示可能な文字列をリストします。

mb-strip

オブジェクト ファイルからシンボルを削除するプログラム。ファイル サイズを削減し、ファイル内のシンボル情報が見られないようにするために使用します。

その他のプログラムおよびファイル

次の Tcl および Tk シェルは、さまざまなフロント エンド プログラムから起動されます。

- cygitclsh30
- cygitkwish30
- cygtclsh80
- cygwish80
- tix4180

その他のリソースおよび法的通知

ザイリンクス リソース

アンサー、資料、ダウンロード、フォーラムなどのサポート リソースは、[ザイリンクス サポート](#) サイトを参照してください。

ソリューション センター

デバイス、ツール、IP のサポートについては、[ザイリンクス ソリューション センター](#)を参照してください。デザイン アシスタント、アドバイザリ、トラブルシューティングのヒントなどが含まれます。

Xilinx Documentation Navigator およびデザイン ハブ

Xilinx Documentation Navigator (DocNav) では、ザイリンクスの資料、ビデオ、サポート リソースにアクセスでき、特定の情報を取得するためにフィルター機能や検索機能を利用できます。DocNav を開くには、次のいずれかを実行します。

- Vivado IDE で [Help] → [Documentation and Tutorials] をクリックします。
- Windows で [スタート] → [すべてのプログラム] → [Xilinx Design Tools] → [DocNav] をクリックします。
- Linux コマンド プロンプトに「docnav」と入力します。

ザイリンクス デザイン ハブには、資料やビデオへのリンクがデザイン タスクおよびトピックごとにまとめられており、これらを参照することでキー コンセプトを学び、よくある質問 (FAQ) を参考に問題を解決できます。デザイン ハブにアクセスするには、次のいずれかを実行します。

- DocNav で [Design Hubs View] タブをクリックします。
- ザイリンクス ウェブサイトの[デザイン ハブ](#) ページを参照します。

注記: DocNav の詳細は、ザイリンクス ウェブサイトの [Documentation Navigator](#) ページを参照してください。



注意: DocNav からは、日本語版は参照できません。ウェブサイトのデザイン ハブ ページをご利用ください。

参考資料

このガイドでは、次の Vivado® Design Suite ガイドが参照されています。

注記: 日本語版のバージョンは、英語版より古い場合があります。

1. SDK ヘルプ (UG782)
2. 『Zynq-7000 SoC テクニカル リファレンス マニュアル』(UG585: [英語版](#)、[日本語版](#))
3. 『MicroBlaze プロセッサ リファレンス ガイド』(UG081: [英語版](#)、[日本語版](#))
4. 『Zynq-7000 SoC: エンベデッド デザイン チュートリアル』(UG1165)

その他のザイリンクス資料

5. 『Vivado Design Suite ユーザー ガイド: エンベデッド プロセッサ ハードウェア デザイン』(UG898)
6. 『Vivado Design Suite チュートリアル: エンベデッド プロセッサ ハードウェア デザイン』(UG940)
7. 『基本的なソフトウェア プラットフォームの生成リファレンス ガイド』(UG1138)
8. 『Zynq-7000 SoC ソフトウェア開発者向けガイド』(UG821: [英語版](#)、[日本語版](#))
9. 『Zynq UltraScale+ MPSoC パッケージおよびピン配置ユーザー ガイド』(UG1075: [英語版](#)、[日本語版](#))
10. 『Zynq UltraScale+ MPSoC テクニカル リファレンス マニュアル』(UG1085: [英語版](#)、[日本語版](#))
11. 『Zynq UltraScale+ MPSoC ソフトウェア開発者向けガイド』(UG1137: [英語版](#)、[日本語版](#))
12. 『Zynq UltraScale+ MPSoC クイック エミュレーター ユーザー ガイド』(UG1169)
13. 『Zynq デバイスの OpenAMP フレームワーク: 入門ガイド』(UG1186)

その他のリソース

14. GNU のウェブ サイト: <http://www.gnu.org>
15. Red Hat Insight のウェブ サイト: <http://sources.redhat.com/insight>.

トレーニング リソース

ザイリンクスでは、このガイドに含まれるコンセプトを説明するさまざまなトレーニング コースおよび QuickTake ビデオを提供しています。次のリンクから関連するトレーニング リソースを参照してください。

1. [Zynq-7000 SoC QuickTake ビデオ: Zynq 開発ツールの概要](#)
2. [Zynq-7000 SoC QuickTake ビデオ: ザイリンクス SDK のシステム パフォーマンス ツールの概要](#)
3. [Zynq-7000 SoC QuickTake ビデオ: ザイリンクス SDK を使用して 5 分で「Hello World」を作成](#)
4. [Zynq-7000 SoC QuickTake ビデオ: ザイリンクス SDK を使用した Zynq ベアメタル アプリケーション開発](#)

お読みください: 重要な法的通知

本通知に基づいて貴殿または貴社 (本通知の被通知者が個人の場合には「貴殿」、法人その他の団体の場合には「貴社」。以下同じ) に開示される情報 (以下「本情報」といいます) は、ザイリンクスの製品を選択および使用することのためにのみ提供されます。適用される法律が許容する最大限の範囲で、(1) 本情報は「現状有姿」、およびすべて受領者の責任で (with all faults) という状態で提供され、ザイリンクスは、本通知をもって、明示、黙示、法定を問わず (商品性、非侵害、特定目的適合性の保証を含みますがこれらに限られません)、すべての保証および条件を負わない (否認する) ものとし、また、(2) ザイリンクスは、本情報 (貴殿または貴社による本情報の使用を含む) に関し、起因し、関連する、いかなる種類・性質の損失または損害についても、責任を負わない (契約上、不法行為上 (過失の場合を含む)、その他のいかなる責任の法理によるかを問わない) ものとし、当該損失または損害には、直接、間接、特別、付随的、結果的な損失または損害 (第三者が起こした行為の結果被った、データ、利益、業務上の信用の損失、その他あらゆる種類の損失や損害を含みます) が含まれるものとし、それは、たとえ当該損害や損失が合理的に予見可能であったり、ザイリンクスがそれらの可能性について助言を受けていた場合であったとしても同様です。ザイリンクスは、本情報に含まれるいかなる誤りも訂正する義務を負わず、本情報または製品仕様のアップデートを貴殿または貴社に知らせる義務も負いません。事前の書面による同意のない限り、貴殿または貴社は本情報を再生産、変更、頒布、または公に展示してはなりません。一定の製品は、ザイリンクスの限定的保証の諸条件に従うこととなるので、<http://japan.xilinx.com/legal.htm#tos> で見られるザイリンクスの販売条件を参照してください。IP コアは、ザイリンクスが貴殿または貴社に付与したライセンスに含まれる保証と補助的条件に従うことになります。ザイリンクスの製品は、フェイルセーフとして、または、フェイルセーフの動作を要求するアプリケーションに使用するために、設計されたり意図されたりしていません。そのような重大なアプリケーションにザイリンクスの製品を使用する場合のリスクと責任は、貴殿または貴社が単独で負うものです。<http://japan.xilinx.com/legal.htm#tos> で見られるザイリンクスの販売条件を参照してください。

自動車用のアプリケーションの免責条項

オートモーティブ製品 (製品番号に「XA」が含まれる) は、ISO 26262 自動車用機能安全規格に従った安全コンセプトまたは余剰性の機能 (「セーフティ設計」) がない限り、エアバッグの展開における使用または車両の制御に影響するアプリケーション (「セーフティ アプリケーション」) における使用は保証されていません。顧客は、製品を組み込むすべてのシステムについて、その使用前または提供前に安全を目的として十分なテストを行うものとします。セーフティ設計なしにセーフティ アプリケーションで製品を使用するリスクはすべて顧客が負い、製品の責任の制限を規定する適用法令および規則にのみ従うものとします。

© Copyright 2016 Xilinx, Inc. Xilinx, Xilinx のロゴ、Artix、ISE、Kintex、Spartan、Virtex、Vivado、Zynq、およびこの文書に含まれるその他の指定されたブランドは、米国およびその他の各国のザイリンクス社の商標です。すべてのその他の商標は、それぞれの保有者に帰属します。

この資料に関するフィードバックおよびリンクなどの問題につきましては、jpn_trans_feedback@xilinx.com まで、または各ページの右下にある [フィードバック送信] ボタンをクリックすると表示されるフォームからお知らせください。フィードバックは日本語で入力可能です。いただきましたご意見を参考に早急に対応させていただきます。なお、このメール アドレスへのお問い合わせは受け付けておりません。あらかじめご了承ください。