

XST ユーザー ガイド (Virtex-6、Spartan-6、 7 シリーズ デバイス用)

UG687 (v14.3) 2012 年 10 月 16 日

該当するソフトウェア バージョン : ISE Design Suite 14.3 および 14.4



Notice of Disclaimer

The information disclosed to you hereunder (the “Materials”) is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available “AS IS” and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of the Limited Warranties which can be viewed at <http://www.xilinx.com/warranty.htm>; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in Critical Applications: <http://www.xilinx.com/warranty.htm#critapps>.

© Copyright 2002–2012 Xilinx Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. The PowerPC name and logo are registered trademarks of IBM Corp., and used under license. All other trademarks are the property of their respective owners.

本資料は英語版 (v 14.3) 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。

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 までお知らせください。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。

改訂履歴

日付	バージョン	
2012 年 10 月 16 日	14.3	変更リクエストに基づいて、次を変更： <ul style="list-style-type: none">・ run コマンドを複数使用した場合の警告について「run コマンド」セクションに追加・ 「HDL コーディング手法」の章の Log2 関数のコードを変更・ 「非対称ポートのサポート (ブロックRAM)」セクションに、データ バスが 18 ビットを超える場合は RAMB36E1 ブロックが必要であることを示す警告を追加・ VHDL 浮動小数点と実数演算パッケージに関するセクションを削除

目次

改訂履歴	2
1: はじめに	9
アーキテクチャ サポート	9
コード例	9
構文例	9
略語	9
その他のリソース	10
2: XST プロジェクトの作成および合成	11
HDL 合成プロジェクトの作成	11
ISE Design Suite での XST の実行	13
XST をコマンド ライン モードで実行	13
3: VHDL のサポート	25
VHDL の IEEE サポート	25
VHDL のデータ型	26
VHDL のオブジェクト	32
VHDL の演算子	33
VHDL のエンティティとアーキテクチャの記述	34
VHDL の組み合わせ回路	46
VHDL の順序ロジック	58
VHDL の関数とプロシージャ	64
VHDL のアサート文	66
VHDL ライブラリおよびパッケージ	70
VHDL のファイル タイプ サポート	74
VHDL 構文	80
VHDL の予約語	84
4: Verilog サポート	85
Verilog デザイン	85
Verilog の機能	86
詳細情報	86
Verilog-2001 のサポート	87
Verilog の変数による部分的ビット選択	88
Verilog 構造記述	89
Verilog パラメーター	92

Verilog パラメーターと属性の競合	95
Verilog の使用制限.....	96
Verilog -2001 の属性とメタ コメント.....	99
Verilog 構文.....	101
Verilog のシステム タスクおよび関数	104
Verilog プリミティブ	108
Verilog ユーザー定義プリミティブ (UDP)	108
Verilog の予約語.....	111
5 : Verilog ビヘイビア記述.....	113
Verilog ビヘイビア記述の変数	113
初期値	114
reg および wire の配列	114
多次元配列	115
データ型.....	115
使用可能な文	117
論理式	117
ブロック	120
モジュール	121
継続代入文.....	123
手続き代入文.....	124
タスクおよび関数	134
ブロッキングおよびノンブロッキング手続き代入文	138
定数	138
マクロ.....	139
include ファイル	139
Verilog ビヘイビア記述のコメント	140
generate 文.....	141
6 : 混合言語のサポート.....	145
VHDL と Verilog の混合	145
インスタンス化	145
VHDL および Verilog のライブラリ.....	146
VHDL/Verilog の境界規則	147
ジェネリックのサポート	150
ポート マップ	151
LSO ファイル.....	152
7 : HDL コーディング手法	157
VHDL の利点.....	157

Verilog の利点.....	157
マクロ推論フローの概要	158
フリップフロップおよびレジスタ	159
ラッチ	165
トライステート	168
カウンタおよびアキュムレータ	174
シフトレジスタ	180
ダイナミックシフトレジスタ.....	191
マルチプレクサ	195
四則演算ブロックの HDL コーディング手法.....	203
コンパレータ	209
除算器	211
加算器、減算器、加減算器.....	213
乗算器	218
乗加算と乗累算	224
DSP の推論	229
リソース共有	233
RAM の HDL コード記述方法	236
ROM の HDL コード記述方法.....	305
FSM のコンポーネント.....	313
ブラックボックス.....	328
8: FPGA の最適化.....	333
ブロック RAM へのロジックのマッピング	333
フリップフロップのインプリメンテーション ガイドライン	334
フリップフロップのリタイミング	336
エリア制約を設定した場合のスピード最適化	337
インプリメンテーション制約	339
デバイス プリミティブのサポート.....	340
UniMacro ライブラリの使用	347
コアの処理	348
LUT へのロジックのマッピング	350
デバイス上の配置の制御	352
バッファの挿入	354
XST での PCI フローの使用	354
9: デザイン制約.....	357
制約の指定	357
制約の優先順序	358

合成オプションの設定	359
VHDL の属性	360
Verilog-2001 の属性	362
XCF (ザイリンクス制約ファイル)	365
10 : 一般制約	369
I/O バッファの追加	370
ボックス タイプ	371
バス区切り文字	373
大文字/小文字の区別	374
Case 文のインプリメンテーション形式	375
複製接尾語の設定	377
フル ケース	379
RTL 回路図の生成	381
ジェネリック	382
HDL ライブラリ マップ ファイル	385
階層区切り文字	387
合成制約ファイルの無視	389
I/O 規格	390
キープ	391
階層の維持	393
ライブラリ検索順	396
LOC	397
ネットリスト階層	398
最適化エフォート	400
最適化目標	402
パラレル ケース	403
RLOC	404
保存	405
合成制約ファイル	406
Translate Off と Translate On	407
Verilog インクルード ディレクトリ	408
Verilog マクロ	409
作業ディレクトリ	411
11 : HDL 制約	413
FSM 自動抽出	414
列挙型エンコード手法	416
等価レジスタの削除	417

FSM エンコード方法の指定	419
MUX の最小サイズ	421
リソース共有	423
セーフ インプリメンテーション	425
セーフ リカバリ ステート	428
12 : FPGA 制約 (タイミング制約以外)	429
非同期から同期への変換	430
自動 BRAM パッキング	432
ブロック RAM Read-First インプリメンテーション	433
BRAM 使用率	436
バッファ タイプ (BUFFER_TYPE)	439
トライステートからロジックへの変換	440
コアの検索ディレクトリ	442
DSP の使用率	443
BUFGCE の抽出	445
FSM スタイル	446
LUT の結合	448
単一 LUT へのエンティティのマップ	449
BRAM へのロジックへのマップ	451
最大ファンアウト	453
最初のステージの移動	456
最後のステージの移動	459
乗算器スタイル	462
グローバル クロック バッファの数	464
インスタンス化されたプリミティブの最適化	465
I/O レジスタの IOB 内へのパック	467
電力削減	468
RAM の抽出	470
RAM スタイル	472
コアの読み込み	475
制御セットの削減	477
レジスタ自動調整	478
レジスタの複製	483
ROM の抽出	485
ROM スタイル	487
シフトレジスタの抽出	489
シフトレジスタの最小サイズ	491
スライス (LUT-FF ペア) 使用率	493

スライス (LUT-FF ペア) 使用率の許容範囲	495
キャリー チェーンの使用	497
クロック イネーブルの使用	499
DSP ブロックの使用	502
ロー スキュー ラインの使用	505
同期セットの使用	506
同期リセットの使用	508
13 : タイミング制約	511
タイミング制約の適用	511
クロック信号	514
クロス クロック解析	515
FROM-TO	516
グローバル最適化目標	517
オフセット	520
周期	521
タイミング名	522
ネットのタイミング名	523
タイムグループ	524
タイミング無視	525
タイミング制約の書き込み	526
14 : サードパーティの制約	527
VHDL でのサードパーティ制約	527
Verilog でのサードパーティ制約	527
サードパーティの制約と同等の XST 制約	528
15 : 合成レポート	533
合成レポートの内容	533
合成レポートのナビゲート	542
合成レポートの情報	542
16 : 命名規則	545
命名規則のコード例	545
ネットの命名規則	551
インスタンス命名規則	552
大文字/小文字の保持	552
命名規則の制御方法	553
付録 その他のリソース	555

はじめに

アーキテクチャ サポート

このガイドは、ザイリンクスの Virtex®-6、Spartan®-6 および 7 シリーズ デバイス用です。特に記述のない限り、このガイドのすべての機能および制約でこれらのデバイスがサポートされます。その他のデバイスについては、『XST ユーザー ガイド (Virtex-4、Virtex-5、Spartan-3、および CPLD デバイス用)』を参照してください。

コード例

このガイドのコード例は、本書が作成された時点のものです。アップデートおよびその他の例はftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip からダウンロードしてください。各ディレクトリには、summary.txt が含まれ、簡単な概要と共にすべての例がまとめてリストされています。

構文例

このガイドの構文例では、この制約またはコマンド ライン オプションを特定のツールまたは手法 (VHDL、Verilog、UCF、XCF、ISE® Design Suite、およびコマンド ライン) でどのように設定するかを示しています。すべての属性がどのツールや方法でも指定できるわけではありません。ツールや手法が記述されていない場合は、その方法では制約が設定できないことを示しています。

略語

短縮形	説明
HDL	Hardware Description Language (ハードウェア記述言語)
VHDL	VHSIC Hardware Description Language (VHSIC ハードウェア記述言語)
RTL	Register Transfer Level (レジスタ転送レベル)
LRM	Language Reference Manual (言語リファレンス マニュアル)
FSM	Finite State Machine (有限ステートマシン)
EDIF	Electronic Data Interchange Format (電子データ交換フォーマット)

短縮形	説明
LSO	Library Search Order (ライブラリ検索順)
XST	Xilinx® Synthesis Technology (XST)
XCF	XST Constraint File (XST 制約ファイル)

その他のリソース

XST およびその他のリファレンス マニュアルについては、本書最後の「[その他のリソース](#)」を参照してください。

XST プロジェクトの作成および合成

Xilinx® Synthesis Technology (XST) ツールには、次のような特徴があります。

- ・ ザイリンクス所有のロジック合成ソリューション
- ・ 次から使用できます。
 - ISE® Design Suite
 - PlanAhead™
- ・ コマンド ライン モードでスタンドアロン ツールとして起動可能

XST ソフトウェアには、次のような特徴があります。

1. HDL (VHDL または Verilog) の記述を認識
2. ザイリンクス特有のロジックリソースの合成ネットリストに変換
3. デザインの論理記述である合成済みネットリストは
 - a. デザイン インプリメンテーション ツールのフローで処理されます。
 - b. 物理記述に変換されます。
 - c. ザイリンクス デバイスのプログラム用ビットストリーム ファイルに変換されます。

HDL 合成プロジェクトの作成

XST では、デザインに関する情報とその処理方法に関する情報が分けて保存されます。

ファイル	内容
HDL 合成プロジェクト ファイル	デザインに関する情報
XST のスクリプト ファイル	合成パラメーター

HDL 合成プロジェクト ファイルの定義

HDL 合成プロジェクト ファイルには、次の特徴があります。

- ・ ASCII テキスト ファイル
- ・ デザインに含まれる HDL ソース ファイルをリスト
- ・ 1 行ごとに 1 つの HDL ソース ファイルを指定
- ・ 拡張子は通常 .prj

HDL 合成プロジェクト ファイルの構文

```
<hdl_language> <compilation_library> <source_file>
```

- ・ hdl_language
 - HDL ソース ファイルが VHDL か Verilog かを示します。
 - VHDL と Verilog の混合言語プロジェクトの作成を指定できます。
- ・ compilation_library
 - HDL がコンパイルされる論理ライブラリを指定します。
 - デフォルトの論理ライブラリは work です。
- ・ source_file
 - HDL ソース ファイルを指定します。
 - パスは絶対パスでも相対パスでも使用できます。
 - 相対パスは、HDL 合成プロジェクト ファイルのディレクトリに相対的にします。

ISE Design Suite でのサンプル HDL 合成プロジェクト ファイルの作成

ISE® Design Suite でのサンプル HDL 合成プロジェクト ファイルを作成するには、次の手順に従います。

1. 次のコードを実行します。

```
vhdl      work      my_vhdl1.vhd
verilog   work      my_vlg1.v
vhdl      my_vhdl_lib ../my_other_srcdir/my_vhdl2.vhd
verilog   my_vlg_lib my_vlg2.v
```

このコードは、相対パスを使用しています。

2. XST により、プロジェクト ディレクトリに HDL 合成プロジェクト ファイルが作成されます。このファイルの拡張子は .prj です。
3. HDL ソース ファイルをプロジェクトに入力するたびに、新しいファイル名が合成プロジェクト ファイルに追加されます。

詳細は、ISE ヘルプを参照してください。

コマンド ラインからの HDL 合成プロジェクト ファイルの作成

コマンド ラインから HDL 合成プロジェクト ファイルを作成するには、次の手順に従います。

1. HDL 合成プロジェクト ファイルを手動で作成します。
2. **run** コマンドラインに入力ファイル名 (**-ifn**) オプションを入力します。
-ifn オプションで XST に HDL 合成プロジェクト ファイルのディレクトリを認識させます。

ISE Design Suite での XST の実行

ISE® Design Suite で XST を実行するには、次の手順に従います。

1. プロジェクトを新規作成します。
[File] → [New Project]
2. HDL ソース ファイルをインポートします。
[Project] → [Add Copy of Source]
3. 最上位レベル ブロックを選択します。
[Design] → [Hierarchy]
4. ISE Design Suite で正しいブロックが最上位レベル ブロックとして選択されていない場合は、次を実行してください。
 - a. 正しいブロックを選択します。
 - b. [Set as Top Module] を右クリックします。
 - c. [Processes] で [Synthesize - XST] プロセスを右クリックします。
5. 使用可能な合成オプションをすべて表示するには、[Process Properties] を選択します。
6. 合成を開始するには
 - a. 右クリックします。
 - b. [Run] をクリックします。

詳細は、ISE ヘルプを参照してください。

XST をコマンド ライン モードで実行

XST は、次のいずれかの方法でコマンド ラインから実行できます。

- ・ [XST をスタンドアロン ツールとして実行](#)
- ・ [XST をインタラクティブに実行](#)
- ・ [XST をスクリプト モードで実行](#)

XST をスタンドアロン ツールとして実行

XST はスタンドアロン ツールとして実行できます。

XST を ISE® Design Suite のグラフィカル環境を使用するのではなく、スクリプト記述されたデザイン インプリメンテーションの一部として実行する場合は、[コマンド ライン モード](#)を使用します。

環境変数の設定

XST を実行する前に、次の環境変数が正しいインストール ディレクトリを指定するように設定します。この例は、64-bit Linux の場合です。

```
setenv XILINX setenv PATH $XILINX/bin/lin64:$PATH
setenv LD_LIBRARY_PATH $XILINX/lib/lin64:$LD_LIBRARY_PATH
```

XST の起動

OS	コマンド
Windows	xst.exe
Linux	xst

コマンドラインの構文

```
xst[.exe] [-ifn in_file_name] [-ofn out_file_name] [-intstyle]
[-filter msgfilter_file_name]
```

- ・ -ifn
実行するコマンドラインを含む XST スクリプト ファイルを指定します。
 - -ifn オプションが指定されない場合、XST はインタラクティブに実行されます。
 - -ifn が指定される場合、XST はスクリプト モードで実行されます。
- ・ -ofn
XST のログ ファイルを指定したディレクトリおよびファイルに出力できます。デフォルトでは、XST のログは work ディレクトリの .srp ファイルに記述されます。
- ・ -intstyle
標準出力のレポートを制御します。詳細は、「[Silent モード](#)」を参照してください。
- ・ -filter
コマンドライン モードで制限付きのメッセージ フィルターを有効にします。

コマンドライン モードでのメッセージ フィルターの使用

コマンドライン モードで制限付きのメッセージ フィルターを有効にするには

1. メッセージ フィルターなしで 1 度デザインをコマンドライン モードで合成します。
2. ザイリンクスの xreport ツールを実行します。

```
xreport -config example.xreport -reports_dir . -filter
example.filter example &
```

- ・ -reports_dir で定義されるディレクトリは、XST のログ ファイルが作成されるのと同じディレクトリにする必要があります。
- ・ 上記の例では、次が仮定されます。
 - 最初の実行から生成されるログ ファイルには example.srp. という名前が付けられます。
 - ログ ファイルは XST の起動ディレクトリと同じディレクトリにあります。

詳細は、**xreport -h** を実行してください。

3. [Design Overview] → [Summary] をクリックします。
4. [Design Properties] → [Enable Message Filtering] をクリックします。
5. [Design Overview] → [Synthesis Messages] をクリックし、XST ログ ファイルからのメッセージを表示します。
6. フィルターするメッセージを選択します。
7. 右クリックします。
8. [Filter All Instances of This Message] または [Filter This Instance Only] を選択します。

このメッセージフィルター設定は、`example.filter` ファイルに保存されます。

9. コマンドライン モードで **-filter** オプションを使用して XST をもう 1 度実行します。

```
xst ... -filter example.filter ...
```

アドバンス フィルターを設定する場合、または前にオフにしたメッセージを再び表示されるようにするには

1. [Synthesis Messages] ペインを右クリックします。
2. [Edit Message Filters] をクリックします。

XST をインタラクティブに実行

- ・ **-ifn** オプションなしに XST を実行し、コマンドラインに命令を入力します。
- ・ インタラクティブ モードの場合、XST ログ ファイルは作成されないため、**-ifn** オプションは使用しても何の影響もありません。

XST をスクリプト モードで実行

- ・ コマンド プロンプトにコマンドを入力する代わりに、必要なコマンドおよびオプションを含む XST スクリプト ファイルを作成します。
- ・ スクリプトで記述したインプリメンテーション フローで XST を実行する場合、次のいずれかを実行する必要があります。
 - XST スクリプト ファイルを前もって手動で作成
 - XST スクリプト ファイルを生成

XST のスクリプト ファイル

XST のスクリプト ファイルには、次のような特徴があります。

- ・ ASCII テキスト ファイル
- ・ 1 つまたは複数の XST コマンドが含まれます。
- ・ **-ifn** オプションで XST に渡されます。

```
xst -ifn myscript.xst
```

- ・ スクリプト ファイルに必ず使用しなければならないファイル拡張子はありませんが、ISE® Design Suite では拡張子 `.xst` のスクリプト ファイルが作成されます。

XST スクリプト ファイルの可読性の改善

- ・ 1 つの行に 1 つのオプションと値を記述する。
- ・ 最初の行には **run コマンド**のみを指定する (オプションは指定しない)。
- ・ コマンドの最初の行から最後の行までの間に空行を作らない。
- ・ オプションと値の各行はダッシュ (-) で開始する。
- ・ 各オプションごとに 1 つの値を指定する
- ・ 値を省略できるオプションはなし
- ・ オプションの値は、次のいずれかになります。
 - XST で定義されている値 (**yes**、**no** など)
 - 整数
 - 文字列 (ファイル名、最上位のエンティティ名など)
 - ◆ -vlgincdir オプションでは、複数のディレクトリを値として指定できます。
 - ◆ ディレクトリ名はスペースで区切ります。
詳細は、「[コマンドライン モードでのスペースを含む名前](#)」を参照してください。
 - ◆ ディレクトリリストを {} で囲みます。
`-vlgincdir {c:\vlg1 c:\vlg2}`
- ・ # 文字を使用すると、次が実行できます。
 - オプションをコメントアウトします。
 - スクリプト ファイルにさらにコメントを含めます。

XST のスクリプト ファイルの例

```
run
-ifn myproject.prj
-ofn myproject.ngc
-ofmt NGC
-p virtex6
# -opt_mode area
-opt_mode speed
-opt_level 1
```


XST のコマンド

XST では、次のコマンドが認識されます。

- ・ [run コマンド](#)
- ・ [set コマンド](#)
- ・ [help コマンド](#)

run コマンド

run コマンドには、次のような特徴があります。

- ・ 主な合成コマンドです。
- ・ スクリプト ファイルごとに 1 回だけ使用します。
- ・ 合成をそのまま実行します。
 - － HDL ソースファイルを解析すると合成が開始されます。
 - － 最終的なネットリストを生成すると合成は終了します。
- ・ 次のいずれかのために、HDL 解析およびエラボレーションを実行します。
 - － 言語に準拠しているかどうかの検証
 - － HDL ファイルのプリコンパイル

注記： 1 つのスクリプト内での複数の **run** コマンドの使用はサポートされません。

run コマンドの構文

run option_1 value option_2 value ...

- ・ HDL 記述 (たとえば、最上位レベルモジュール) のエレメントに指定されるオプション値を除き、**run** コマンドには大文字/小文字関係なく使用できます。
- ・ オプションは小文字のみか大文字のみで指定できます。たとえば、オプションの値 **yes** と **YES** はどちらも同じように処理されます。

run コマンドの設定

次の表は、**run** コマンドに必要な設定とオプションの設定を示しています。その他の[コマンドライン モード](#)のオプションについては、次を参照してください。

- ・ [第 10 章：一般制約](#)
- ・ [第 11 章：HDL 制約](#)
- ・ [第 12 章：FPGA 制約](#)
- ・ [第 13 章：タイミング制約](#)

run コマンドの設定 (必須)

オプション	コマンドライン名	オプション値	注記
入力ファイル名	-ifn	HDL 合成プロジェクト ファイルへの相対パスまたは絶対パス	
出力ファイル名	-ofn	合成後の NGC ネットリストを保存するファイルへの相対パスまたは絶対パス。	拡張子 .ngc は削除できません。
ターゲット デバイス	-p	<ul style="list-style-type: none"> ・ xc6v1x240t-ff1759-1 のような特定のデバイス、または ・ Virtex®-6 デバイスのようなデバイス ファミリ名 	
最上位モジュール名	-top	デザインの最上位レベルを記述する VHDL エンティティまたは Verilog モジュール名。	コンポーネント インスタンス化をデザイン エンティティおよびアーキテクチャにまとめるのに別のコンフィギュレーション宣言を使用している場合、値はそのコンフィギュレーションの名前にします。

run コマンドの設定 (オプション)

オプション	コマンド ライン名
VHDL Top Level Architecture (最上位レベルの VHDL エンティティに関連付けられた特定の VHDL アーキテクチャ名。デザインの最上位レベルが Verilog で記述されている場合は、使用不可)。	-ent
Optimization Goal (最適化目標)	-opt_mode
Optimization Effort (最適化エフォート)	-opt_level
Power Reduction (電力削減)	-power
Use Synthesis Constraints File (合成制約ファイルの使用)	-iuc
Synthesis Constraints File (合成制約ファイル)	-uc
Keep Hierarchy (階層の維持)	-keep_hierarchy
Netlist Hierarchy (ネットリスト階層)	-netlist_hierarchy
Global Optimization Goal (グローバル最適化目標)	-glob_opt
Generate RTL Schematic (RTL 回路図の生成)	-rtlview
Read Cores (コアの読み出し)	-read_cores
Cores Search Directories (コア検索ディレクトリ)	-sd
Write Timing Constraints (タイミング制約の書き込み)	-write_timing_constraints
Cross Clock Analysis (クロス クロック解析)	-cross_clock_analysis
Hierarchy Separator (階層区切り文字)	-hierarchy_separator
Bus Delimiter (バス区切り文字)	-bus_delimiter
LUT-FF Pairs Utilization Ratio (スライス (LUT-FF ペア) の使用率)	-slice_utilization_ratio
BRAM Utilization Ratio (BRAM 使用率)	-bram_utilization_ratio
DSP Utilization Ratio (DSP 使用率)	-dsp_utilization_ratio
Case (大文字/小文字)	-case
Library Search Order (ライブラリ検索順)	-lso
Verilog Include Directories (Verilog インクルード ディレクトリ)	-vlgincdir
Generics (ジェネリック)	-generics
Verilog Macros (Verilog マクロ)	-define
FSM Extraction (FSM 抽出)	-fsm_extract
FSM Encoding Algorithm (FSM エンコーディング アルゴリズム)	-fsm_encoding
Safe Implementation (セーフ インプリメンテーション)	-safe_implementation
Case Implementation Style (case 文のインプリメント方法)	-vlgcase
FSM Style (FSM スタイル)	-fsm_style
RAM Extraction (RAM 抽出)	-ram_extract
RAM Style (RAM スタイル)	-ram_style
ROM Extraction (ROM 抽出)	-rom_extract
ROM Style (ROM スタイル)	-rom_style

オプション	コマンド ライン名
Automatic BRAM Packing (自動 BRAM パッキング)	-auto_bram_packing
Shift Register Extraction (シフトレジスタ抽出)	-shreg_extract
Shift Register Minimum Size (シフトレジスタの最小サイズ)	-shreg_min_size
Resource Sharing (リソース共有)	-resource_sharing
Use DSP Block (DSP ブロックの使用)	-use_dsp48
Asynchronous To Synchronous (非同期から同期への変換)	-async_to_sync
Add I/O Buffers (I/O バッファの追加)	-iobuf
Max Fanout (最大ファンアウト)	-max_fanout
Number of Clock Buffers (クロック バッファ数)	-bufg
Register Duplication (レジスタの複製)	-register_duplication
Equivalent Register Removal (等価レジスタの削除)	-equivalent_register_removal
Register Balancing (レジスタの自動調整)	-register_balancing
Move First Flip-Flop Stage (最初のフリップフロップ ステージの移動)	-move_first_stage
Move Last Flip-Flop Stage (最後のフリップフロップ ステージの移動)	-move_last_stage
Pack I/O Registers into IOBs (I/O レジスタの IOB へのパック)	-iob
LUT Combining (LUT の結合)	-lc
Reduce Control Sets (制御セットの削減)	-reduce_control_sets
Use Clock Enable (クロック イネーブルの使用)	-use_clock_enable
Use Synchronous Set (同期セットの使用)	-use_sync_set
Use Synchronous Reset (同期リセットの使用)	-use_sync_reset
Optimize Instantiated Primitives (インスタンス化されたプリミティブの最適化)	-optimize_primitives

set コマンド

set コマンドは、**run コマンド** を実行する前にプリファレンスを設定するために使用します。

set -option_name [option_value]

詳細は、第 9 章「デザイン制約」を参照してください。

set コマンドのオプション

オプション	説明	値
-tmpdir	現在のセッションで XST により生成された一時ファイルを格納するディレクトリへのパスを指定	ディレクトリへのパス
-xsthdmdir	作業ディレクトリ (HDL コンパイルで生成されたファイルのディレクトリ) を指定	ディレクトリへのパス
-xsthdpini	HDL ライブラリ マップ ファイル (INI ファイル)	file_name

help コマンド

help コマンドを使用すると、次が表示されます。

- ・ サポートされるファミリ
- ・ 特定デバイスの全コマンド
- ・ 特定デバイスの特定コマンド

サポートされるファミリ

サポートされるデバイスをすべてリストするには、次を実行してください。

1. 引数を指定せずに **help** と入力します。

help

2. XST で次のメッセージが表示されます。

```
--> help ERROR:Xst:1356 - Help : Missing "-arch ". Please
specify what family you want to target available families:
spartan6 virtex6
```

3. サポートされるファミリのリストは、「available families」に記述されます。

特定デバイスの全コマンド

特定デバイスの全コマンドを表示するには、次の手順に従います。

1. 次をコマンドラインに入力します。

help -arch family_name

family_name にはサポートされるデバイス ファミリ名を指定します。

2. たとえば、Virtex®-6 デバイスのコマンドをすべてリストする場合は、次のように入力します。

```
help -arch virtex6
```

特定デバイスの特定コマンド

特定デバイスの特定コマンドを表示するには、次の手順に従います。

1. 次をコマンド ラインに入力します。

```
help -arch family_name -command command_name
```

- ・ *family_name* にはサポートされるデバイス ファミリ名を指定します。
- ・ *command_name* には、次のいずれかのコマンドを入力します。
 - run
 - set
 - time

2. たとえば、Virtex-6 デバイスの **run** コマンドに関する情報を表示する場合は、次のように入力します。

```
help -arch virtex6 -command run
```

コマンド ライン モードでのスペースを含む名前

XST の **コマンド ライン モード**では、スペースを含むファイル名またはディレクトリ名がサポートされます。

- ・ ファイル名またはディレクトリ名にスペースが含まれる場合は、次の例のように名前を二重引用符 (" ") で囲む必要があります。

```
"C:\my project"
```

- ・ 複数ディレクトリをサポートするオプション (-sd、-vlgincdir) では、ディレクトリを { } で囲みます。

```
-vlgincdir {"C:\my project" C:\temp}
```

- ・ 古いバージョンの XST では、複数のディレクトリを二重引用符 (" ") で囲んでいました。この構文は、スペースを含まないディレクトリ名に対しては、現在のバージョンの XST でもサポートされていますが、既存のスクリプトの複数ディレクトリは新しい構文の { } で囲むようにに変更することをお勧めします。

出力ファイル

XST の出力ファイルは、次のとおりです。

- ・ **通常**の出力ファイル
- ・ **一時**出力ファイル

通常の実出力ファイル

XST では、通常次の出力ファイルが生成されます。

- ・ 出力 NGC ネットリスト (.ngc)
 - ISE® Design Suite では、プロジェクト ディレクトリに NGC ファイルが作成されます。
 - コマンドライン モードの場合、NGC ファイルは次のディレクトリに作成されます。
 - ◆ 現在のディレクトリまたは
 - ◆ `run -ofn` で指定したその他のディレクトリ
- ・ RTL Viewer 用の Register Transfer Level (RTL) ネットリスト (.ngr)
- ・ 合成ログ ファイル (.srp)

一時出力ファイル

- ・ XST の一時ディレクトリには一時的なファイルが生成されます。
- ・ HDL コンパイル ファイルは、この一時ディレクトリに生成されます。
- ・ デフォルトの一時ディレクトリは、現在のディレクトリの下にある xst ディレクトリです。

一時ディレクトリ

システム	ディレクトリ
ワークステーション	/tmp
Windows	TEMP または TMP 環境変数で指定されたディレクトリ

一時ディレクトリの変更

一時ディレクトリを変更するには、次のいずれかで `set -tmpdir <directory>` を実行します。

- ・ XST プロンプト
- ・ XST のスクリプト ファイル

一時ディレクトリの維持

- ・ 一時ディレクトリには、すべての XST セッションの VHDL および Verilog ファイルのコンパイルにより生成されたファイルが含まれます。
- ・ 一時ディレクトリに格納されるファイルの数が増えると、CPU の動作に悪影響を及ぼす場合があります。
- ・ XST ではこの 一時ディレクトリを自動的にクリーンアップしないので、XST の一時ディレクトリのファイルは手動で定期的に削除するようにしてください。

VHDL のサポート

XST では、特に記述のない限り VHDL がサポートされます。

- ・ VHDL は複雑なロジックを簡潔に表現できるハードウェア記述言語です。
- ・ VHDL では、次が実行可能です。
 - 次のシステム構造を記述できます。
 - ◆ システムをサブシステムに分割する方法
 - ◆ 分割されたサブシステムを相互接続する方法
 - 使い慣れたプログラミング言語形式でシステムの機能を指定できます。
 - インプリメンテーションおよびハードウェアへのプログラム前にシステム デザインをシミュレーションできます。
 - 抽象記述を合成して、デバイス特定の詳細なデザインを作成する方法を提供します。

詳細は、次を参照してください。

- ・ IEEE の『VHDL Language Reference Manual』(LRM)
- ・ [第 9 章「デザイン制約」](#)、特に「VHDL 属性」

VHDL の IEEE サポート

XST には、VHDL IEEE 1076-1993 に準拠する解析およびエラボレーション エンジンが含まれます。

XST では、次のような LRM に準拠しない構文もサポートされます。

- ・ 合成またはシミュレーション ツールのほとんどでサポートされる
- ・ コードをかなり単純化できる構文
- ・ 合成に悪影響を及ぼさない
- ・ 結果に悪影響を及ぼさない

LRM に準拠しない例

- ・ 次の場合、LRM ではポート マップを使用してインスタンス化ができません。
 - フォーマル ポートが buffer で、さらに
 - 有効なポートが out の場合
- ・ XST では、この LRM に準拠しない構文もサポートされます。この構文は、上記の LRM に準拠しない構文の条件に当てはまります。

VHDL のデータ型

次に説明する VHDL データ型には、定義済みのパッケージの一部であるものもあります。

コンパイルされる箇所、およびそれらの読み込み方法については、「[VHDL の定義済みパッケージ](#)」を参照してください。

VHDL でサポートされないデータ型

VHDL では、標準パッケージの **real** 型は、ジェネリック値などの計算を実行する目的にのみ使用できます。

real 型の合成可能なオブジェクトタイプは定義できません。

VHDL でサポートされるデータ型

VHDL では、次のデータ型がサポートされます。

- ・ [定義済み列挙型](#)
- ・ [ユーザー定義の列挙型](#)
- ・ [ビット ベクター型](#)
- ・ [整数型](#)
- ・ [多次元配列型](#)
- ・ [VHDL のレコード型](#)

定義済み列挙型

XST では、ハードウェア記述で次の定義済み VHDL 列挙型がサポートされます。

- ・ 標準パッケージで定義される **bit** 型。
使用できる値は、0 (ロジック 0) および 1 (ロジック 1) です。
- ・ 標準パッケージで定義される **boolean** 型。
使用できる属性値は、false または true です。
- ・ IEEE **std_logic_1164** パッケージで定義される **std_logic** 型。
使用できる値については、「std_logic に使用可能な値」の表を参照してください。

この情報は、次の表にまとめられています。

定義済み VHDL 列挙型のサマリ

列挙型	定義されるパッケージ	値
bit	標準パッケージ	<ul style="list-style-type: none"> ・ 論理値 0 ・ 論理値 1
boolean	標準パッケージ	<ul style="list-style-type: none"> ・ false ・ true
std_logic	IEEE std_logic_1164 パッケージ	詳細は、「std_logic に使用可能な値」の表を参照してください。

std_logic に使用可能な値

値	説明	XST での変換
U	初期化なし	XSTでは使用できません
X	不明	ドント ケアとして処理されます
0	low	ロジック 0 として処理されます
1	high	ロジック 1 として処理されます
Z	ハイ インピーダンス	ハイ インピーダンスとして処理されます
W	ウィークな不明状態	XSTでは使用できません
L	ウィーク Low	0 と同じように処理されます
H	ウィーク High	1 と同じように処理されます
-	ドントケア	ドント ケアとして処理されます

XST でサポートされるオーバーロード列挙型

データ型	IEEE パッケージ での定義	サブタイプ	含まれる値
std_ulogic	std_logic_1164	なし	<ul style="list-style-type: none"> std_logic と同じ 9 つの値 定義済みレゾリューション関数を含まない
X01	std_logic_1164	std_ulogic	X、0、1
X01Z	std_logic_1164	std_ulogic	X、0、1、Z
UX01	std_logic_1164	std_ulogic	U、X、0、1
UX01Z	std_logic_1164	std_ulogic	U、X、0、Z

VHDL のユーザー定義の列挙型

独自の列挙型も作成できます。

ユーザー定義の列挙型では、通常有限ステートマシン (FSM) のステートを記述します。

ユーザー定義の列挙型の VHDL コード例

```
type STATES is (START, IDLE, STATE1, STATE2, STATE3) ;
```

VHDL のビット ベクター型

ビット ベクター型

データ型	定義されるパッケージ	モデル
bit_vector	規格	ビット エLEMENTのベクター
std_logic_vector	IEEE std_logic_1164	std_logic エLEMENTのベクター

サポートされる VHDL のオーバーロード型

データ型	IEEE パッケージでの定義
std_ulogic_vector	std_logic_1164
unsigned	std_logic_arith
signed	std_logic_arith

VHDL の整数型

整数型は、定義済みの VHDL 型です。

- ・ デフォルトでは、integer は で 32 ビットにインプリメントされます。
- ・ 適用可能な値の範囲をはっきりと定義することで、次のようにインプリメンテーションをさらにコンパクトにできます。

```
type MSB is range 8 to 15
```
- ・ また、整数型をオーバーロードすることで、この定義済みの自然数および正の型の利点を利用することもできます。

VHDL の多次元配列型

XST では、VHDL の多次元配列型がサポートされます。

- ・ 次元数に制限はありませんが、3 次元を超えないように記述することをお勧めします。
- ・ 記述する可能性のある多次元配列のオブジェクトは、次のとおりです。
 - 信号
 - 定数
 - 変数
- ・ 多次元配列型のオブジェクトは、
 - 関数に渡されます。
 - コンポーネント インスタンスエーションで使用されます。

完全に制約が付いた配列型のコード例

配列型はすべての次元で完全に制約を付ける必要があります。

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);  
type TAB12 is array (11 downto 0) of WORD8;  
type TAB03 is array (2 downto 0) of TAB12;
```

マトリックスとして宣言された配列のコード例

配列はマトリックスとして宣言できます。

```
subtype TAB13 is array (7 downto 0, 4 downto 0) of STD_LOGIC_VECTOR (8 downto 0);
```

多次元配列の信号および変数のコード例

次に、代入文における多次元配列の信号および変数の使用例を示します。

1. 次のように宣言したとします。

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);
type TAB05 is array (4 downto 0) of WORD8;
type TAB03 is array (2 downto 0) of TAB05;
signal WORD_A : WORD8;
signal TAB_A, TAB_B : TAB05;
signal TAB_C, TAB_D : TAB03;
constant CNST_A : TAB03 := (
  ("00000000", "01000001", "01000010", "10000011", "00001100"),
  ("00100000", "00100001", "00101010", "10100011", "00101100"),
  ("01000010", "01000010", "01000100", "01000111", "01000100"));
```

2. これで次を指定できるようになります。

- ・ 多次元配列の信号または変数

```
TAB_A <= TAB_B; TAB_C <= TAB_D; TAB_C <= CNST_A;
```

- ・ 1 配列のインデックス

```
TAB_A (5) <= WORD_A; TAB_C (1) <= TAB_A;
```

- ・ 最大の次元数のインデックス

```
TAB_A (5) (0) <= '1'; TAB_C (2) (5) (0) <= '0'
```

- ・ 最初の配列のスライス

```
TAB_A (4 downto 1) <= TAB_B (3 downto 0);
```

- ・ 高次元配列のインデックスと低次元配列のスライス

```
TAB_C (2) (5) (3 downto 0) <= TAB_B (3) (4 downto 1); TAB_D (0) (4) (2 downto 0)
\\ <= CNST_A (5 downto 3)
```

3. 次の宣言文を追加します。

```
subtype MATRIX15 is array(4 downto 0, 2 downto 0) of STD_LOGIC_VECTOR (7 downto 0);
signal MATRIX_A : MATRIX15;
```

4. これで次を指定できるようになります。

- ・ 多次元配列の信号または変数

```
MATRIXA <= CNST_A
```

- ・ 1 行の配列のインデックス

```
MATRIXA (5) <= TAB_A;
```

- ・ 最大の次元数のインデックス

```
MATRIXA (5,0) (0) <= '1';
```

インデックスは、変数にできます。

VHDL のレコード型

```
type mytype is record
  field1 : std_logic;
  field2 : std_logic_vector (3 downto 0)
end record;
```

- ・ レコード型のフィールドは、**record** にもできます。
- ・ 定数をレコード型にできます。
- ・ レコード型に属性を含むことはできません。
- ・ レコード信号への集合代入文がサポートされます。

VHDL のオブジェクト

XST では、次の VHDL のオブジェクトがサポートされます。

- ・ [VHDL の信号](#)
- ・ [VHDL の変数](#)
- ・ [VHDL の定数](#)

VHDL の信号

VHDL 信号は、次で宣言できます。

- ・ アーキテクチャ宣言部分
そのアーキテクチャ内のどこかで VHDL 信号を使用します。
- ・ ブロック
そのブロック内で VHDL 信号を使用します。

VHDL 信号は代入演算子「<=」を使用して代入します。

```
signal sig1 : std_logic;  
sig1 <= '1';
```

VHDL の変数

VHDL の変数は、次のように使用されます。

- ・ process または subprogram 文で宣言します。
- ・ process または subprogram 文内で使用されます。
- ・ 代入演算子「:=」を使用して代入します。

```
variable var1 : std_logic_vector (7 downto 0); var1 := "01010011";
```

VHDL の定数

VHDL 定数は、宣言領域で宣言できます。

- ・ その領域内で使用できます。
- ・ 宣言後、その値は変更できません。

```
signal sig1 : std_logic_vector (5 downto 0); constant init0 :  
std_logic_vector (5 downto 0) := "010111"; sig1 <= init0;
```


VHDL の演算子

XST では、VHDL の演算子がサポートされます。詳細は、「[VHDL 演算子のサポート](#)」を参照してください。

シフト演算子の例

演算子	例	論理上同一
SLL (Shift Left Logic)	sig1 <= A(4 downto 0) sll 2	sig1 <= A(2 downto 0) & "00";
SRL (Shift Right Logic)	sig1 <= A(4 downto 0) srl 2	sig1 <= "00" & A(4 downto 2);
SLA (Shift Left Arithmetic)	sig1 <= A(4 downto 0) sla 2	sig1 <= A(2 downto 0) & A(0) & A(0);
SRA (Shift Right Arithmetic)	sig1 <= A(4 downto 0) sra 2	sig1 <= A(4) & A(4) & A(4 downto 2);
ROL (Rotate Left)	sig1 <= A(4 downto 0) rol 2	sig1 <= A(2 downto 0) & A(4 downto 3);
ROR (Rotate Right)	A(4 downto 0) ror 2	sig1 <= A(1 downto 0) & A(4 downto 2);

VHDL のエンティティとアーキテクチャの記述

VHDL のエンティティとアーキテクチャの記述には、次が含まれます。

- ・ [VHDL の回路記述](#)
- ・ [VHDL のエンティティ宣言](#)
- ・ [VHDL のアーキテクチャ宣言](#)
- ・ [VHDL のコンポーネント インスタンス化](#)
- ・ [VHDL の再帰的なコンポーネント インスタンス化文](#)
- ・ [VHDL のコンポーネント コンフィギュレーション文](#)
- ・ [VHDL のジェネリック](#)

VHDL の回路記述

VHDL の回路記述は、次で構成されています。

- ・ エンティティ宣言
 - 回路の外部表示を提供
 - I/O ポートおよびジェネリックなどの回路のインターフェイスを含め、外側から見えるオブジェクトについて記述
- ・ アーキテクチャ
 - 回路の内部表示を提供
 - 回路ビヘイビアまたはストラクチャを記述

VHDL のエンティティ宣言

回路の I/O ポートはエンティティで宣言します。

各ポートには、次が含まれます。

- ・ name
- ・ mode
 - in
 - out
 - inout
 - buffer
- ・ type

制約の付いたポートと付いていないポート

ポートには、制約を付けたり、付けなかったりできます。

- ・ ポートには、通常は制約が付きます。
- ・ ポートは、エンティティ宣言部分では制約を付けないままにできます。
- ・ 制約を付けない場合、ポートの幅はフォーマル ポートと実際の信号間が接続されるときに、インスタンス化で定義されます。
- ・ 制約の付いていないポートを使用すると、異なるポート幅を定義できるので、同じエンティティのさまざまなインスタンス化を作成できます。
- ・ ザイリンクスでは、次を推奨しています。
 - 制約の付いていないポートは使用しないでください。
 - ジェネリックを使用して制約を付けたポートを定義します。
 - インスタンス化時にこれらのジェネリックに異なる値を適用します。
 - 最上位レベルのエンティティには制約の付いていないポートは含めないでください。
- ・ ポートには、1 次元以上の配列型は使用できません。
- ・ エンティティ宣言でも **VHDL ジェネリック**を宣言できます。

バッファポート モード

バッファポート モードは使用しないようにしてください。

- ・ VHDL では信号が次のように使用される場合にバッファポート モードを使用できます。
 - 内部ポート
 - 出力ポート (内部ドライバーが 1 つしかない)
- ・ バッファポート :
 - 合成中にエラーになる可能性がある
 - シミュレーションからの合成後の結果を有効にするのが難しい

バッファポート モードを使用したコード例 (推奨されない)

```
entity alu is
    port (
        CLK : in  STD_LOGIC;
        A   : in  STD_LOGIC_VECTOR(3 downto 0);
        B   : in  STD_LOGIC_VECTOR(3 downto 0);
        C   : buffer STD_LOGIC_VECTOR(3 downto 0));
end alu;

architecture behavioral of alu is
begin
    process begin
        if rising_edge(CLK) then
            C <= UNSIGNED(A) + UNSIGNED(B) UNSIGNED(C);
        end if;
    end process;
end behavioral;
```

バッファ モードのドロップ

バッファ ポート モードは使用しないようにしてください。

- ・ 上記のコード例で信号 C は、次のように処理されています。
 - バッファ モードでコード記述されています。
 - 内部および出力ポートの両方を使用しています。
- ・ C に接続可能な階層のレベルもすべてバッファとして宣言する必要がありました。
- ・ バッファ モードをドロップするには :
 1. ダミー信号を挿入
 2. ポート C を出力として宣言

バッファ ポート モードを使用しないコード例 (推奨)

```
entity alu is
    port (
        CLK : in  STD_LOGIC;
        A   : in  STD_LOGIC_VECTOR(3 downto 0);
        B   : in  STD_LOGIC_VECTOR(3 downto 0);
        C   : out STD_LOGIC_VECTOR(3 downto 0));
end alu;

architecture behavioral of alu is
    -- dummy signal
    signal C_INT : STD_LOGIC_VECTOR(3 downto 0);
begin
    C <= C_INT;
    process begin
        if rising_edge(CLK) then
            C_INT <= A and B and C_INT;
        end if;
    end process;
end behavioral;
```

VHDL のアーキテクチャ宣言

内部信号はアーキテクチャ文で宣言できます。

各内部信号には、次が指定されます。

- ・ name
- ・ type

アーキテクチャ宣言の VHDL コード例

```
library IEEE;
use IEEE.std_logic_1164.all;

entity EXAMPLE is
    port (
        A,B,C : in std_logic;
        D,E : out std_logic );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
    signal T : std_logic;
begin
    ...
end ARCHI;
```

VHDL のコンポーネント インスタンス化

コンポーネント インスタンス化を使用すると、デザイン ユニット (コンポーネント) を別のデザイン ユニット内にインスタンス化して、階層構造のデザインを記述できます。

コンポーネント インスタンス化を実行するには

1. インスタンス化する機能を記述したデザイン ユニット (エンティティとアーキテクチャ) を作成します。
2. コンポーネントを親デザイン ユニットのアーキテクチャ文の宣言部分でインスタンス化されるように宣言します。
3. 親デザイン ユニットのアーキテクチャ本体部分でこのコンポーネントをインスタンス化して接続します。
4. コンポーネントのフォーマル ポートを実際の信号と親デザイン ユニットのポートにマップ (接続) します。

コンポーネント インスタンス文の要素

コンポーネント インスタンス化文のメイン エLEMENT は次のとおりです。

- ・ ラベル
インスタンスを識別します。
- ・ 関連リスト
 - 予約語の **port map** キーワードで導入されます。
 - コンポーネントのフォーマル ポートを実際の信号または親デザイン ユニットのポートに接続します。
- ・ オプションの関連リスト
 - 予約語の **generic map** キーワードで導入されます。
 - 実際の値をコンポーネントで定義されるフォーマル ジェネリックに提供します。

XST では、コンポーネント宣言で制約が設定されていないベクタがサポートされます。

コンポーネント インスタンス化の VHDL コード例

次は、4 つの NAND2 コンポーネントから構成される半加算器の構造記述例を示しています。

```
--
-- A simple component instantiation example
-- Involves a component declaration and the component instantiation itself
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/instantiation/instantiation_simple.vhd
--
entity sub is
  generic (
    WIDTH : integer := 4);
  port (
    A,B : in  BIT_VECTOR(WIDTH-1 downto 0);
    O   : out BIT_VECTOR(2*WIDTH-1 downto 0));
end sub;
```

```

architecture archi of sub is
begin
    O <= A & B;
end ARCHI;

entity top is
    generic (
        WIDTH : integer := 2);
    port (
        X, Y : in BIT_VECTOR(WIDTH-1 downto 0);
        Z    : out BIT_VECTOR(2*WIDTH-1 downto 0));
end top;

architecture ARCHI of top is

    component sub -- component declaration
        generic (
            WIDTH : integer := 2);
        port (
            A,B : in  BIT_VECTOR(WIDTH-1 downto 0);
            O   : out BIT_VECTOR(2*WIDTH-1 downto 0));
    end component;

begin

    inst_sub : sub -- component instantiation
        generic map (
            WIDTH => WIDTH
        )
    port map (
        A => X,
        B => Y,
        O => Z
    );

end ARCHI;

```

VHDL の再帰的なコンポーネント インスタンスーション文

XST では、VHDL の再帰的なコンポーネント インスタンスーション文がサポートされます。

- ・ 再帰の直接的なインスタンスーションはサポートされません。
- ・ 再帰呼び出しが永久に実行されるのを防ぐために、繰り返す回数は 64 (デフォルト) に制限されています。
- ・ 回数を変更するには、`-recursion_iteration_limit` オプションを使用します。次の例を参照してください。

再帰的なコンポーネント インスタンスレーションの VHDL コード例

```
--
-- Recursive component instantiation
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/instantiation/instantiation_recursive.vhd
--
library ieee;
use ieee.std_logic_1164.all;
library unisim;
use unisim.vcomponents.all;

entity single_stage is
    generic (
        sh_st: integer:=4);
    port (
        CLK : in std_logic;
        DI : in std_logic;
        DO : out std_logic );
end entity single_stage;

architecture recursive of single_stage is
    component single_stage
        generic (
            sh_st: integer);
        port (
            CLK : in std_logic;
            DI : in std_logic;
            DO : out std_logic );
    end component;
    signal tmp : std_logic;
begin
    GEN_FD_LAST: if sh_st=1 generate
        inst_fd: FD port map (D=>DI, C=>CLK, Q=>DO);
    end generate;
    GEN_FD_INTERM: if sh_st /= 1 generate
        inst_fd: FD port map (D=>DI, C=>CLK, Q=>tmp);
        inst_sstage: single_stage
            generic map (sh_st => sh_st-1)
            port map (DI=>tmp, CLK=>CLK, DO=>DO);
    end generate;
end recursive;
```

VHDL のコンポーネント コンフィギュレーション

コンポーネント コンフィギュレーション文を使用すると、コンポーネントを適切なモデルとリンクできます。

- ・ モデルはエンティティ/アーキテクチャのペアです。
- ・ XST では、アーキテクチャ宣言でのコンポーネント コンフィギュレーション文の使用がサポートされます。

```
for instantiation_list : component_name use
  LibName.entity_Name(Architecture_Name);
```

- ・ 下記の文は次を示しています。
 - すべての NAND2 コンポーネントでエンティティ NAND2 とアーキテクチャ ARCHI が使用される
 - デザイン ユニットが work ライブラリでコンパイルされる

```
For all : NAND2 use entity work.NAND2 (ARCHI);
```
- ・ コンポーネント インスタンス化文にコンフィギュレーション節がない場合 :
 - コンポーネントが同じ名前のエンティティに関連付けられます。
 - 選択されたアーキテクチャが最後にコンパイルされたアーキテクチャに関連付けられます。
- ・ エンティティまたはアーキテクチャがない場合、合成中にブラック ボックスが生成されます。
- ・ **コマンド ライン モード**の場合、コンポーネント インスタンス化をデザイン エンティティおよびアーキテクチャに関連付けるのに専用のコンフィギュレーション宣言を使用することもできます。
- ・ 必須の **run コマンド**の [Top Module Name] (-top) オプションの値は、最上位レベルのエンティティ名ではなく、コンフィギュレーション名です。

VHDL のジェネリック

VHDL のジェネリック :

- ・ Verilog のパラメーターと同等
- ・ 拡張可能なデザインを記述するのに役立ちます。
- ・ コンパクトな VHDL コードを記述するために使用できます。
- ・ 次の機能をパラメーター化できます。
 - バス サイズ
 - デザイン ユニットの特定の再帰的エレメントの量

パラメーター化の例

同じ機能をバス サイズを変えて何度もインスタンス化する場合、次のようにジェネリックを使用してデザイン ユニットを 1 つだけ記述します。詳細は、「ジェネリック パラメーターを使用した VHDL コード例」を参照してください。

ジェネリックの宣言

ジェネリック パラメーターは、エンティティ宣言部分で宣言できます。

- ・ XST では、次を含めたあらゆるタイプのジェネリック パラメーターがサポートされます。
 - integer
 - boolean
 - string
 - real
 - std_logic_vector
- ・ ジェネリックはデフォルトの値で宣言します。

ジェネリック パラメーターを使用した VHDL コード例

```
--
-- VHDL generic parameters example
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/generics/generics_1.vhd
--
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity addern is
    generic (
        width : integer := 8);
    port (
        A,B : in std_logic_vector (width-1 downto 0);
        Y : out std_logic_vector (width-1 downto 0) );
end addern;

architecture bhv of addern is
begin
    Y <= A + B;
end bhv;

Library IEEE;
use IEEE.std_logic_1164.all;

entity top is
    port (
        X, Y, Z : in std_logic_vector (12 downto 0);
        A, B : in std_logic_vector (4 downto 0);
        S :out std_logic_vector (17 downto 0) );
end top;

architecture bhv of top is
    component addern
        generic (width : integer := 8);
```

```
    port (  
        A,B : in std_logic_vector (width-1 downto 0);  
        Y : out std_logic_vector (width-1 downto 0) );  
    end component;  
for all : addern use entity work.addern(bhv);  
  
    signal C1 : std_logic_vector (12 downto 0);  
    signal C2, C3 : std_logic_vector (17 downto 0);  
begin  
    U1 : addern generic map (width=>13) port map (X,Y,C1);  
    C2 <= C1 & A;  
    C3 <= Z & B;  
    U2 : addern generic map (width=>18) port map (C2,C3,S);  
end bhv;
```

VHDL のジェネリックと属性の競合

VHDL のジェネリックと属性間の競合の原因には、次のようなものがあります。

- ・ VHDL ジェネリックおよび属性を HDL ソース コードのインスタンスとコンポーネントの両方に適用できる
および
- ・ 属性を制約ファイルで指定できる

競合の解消

XST では、VHDL のジェネリックと属性間の競合を次のように解消します。

- ・ インスタンス (下位レベル) の指定がコンポーネント (上位レベル) の指定より優先されます。
- ・ ジェネリックと属性が同じインスタンスやコンポーネントに適用できる場合、その属性はジェネリックが指定されている箇所に関係なく、優先されます。

同じ制約を定義するのに両方の方法を使用しないでください。このような場合、XST でそれを示すメッセージが表示されます。

- ・ XCF (XST 制約ファイル) で指定される属性は、VHDL コードで指定される属性またはジェネリックよりも常に優先されます。
- ・ ブロックの定義のセキュリティ属性は、ほかのどの属性またはジェネリックよりも優先されます。

この情報は、次の表にまとめられています。

競合の解消のサマリ

項目	これより優先
インスタンスへの指定 (下位レベル)	コンポーネントへの指定 (高位レベル)
インスタンスまたはコンポーネントに適用される属性	同じインスタンスまたはコンポーネントに適用されるジェネリック
ザイリンクス制約ファイル (XCF) で指定した属性	VHDL コードで指定した属性またはジェネリック
ブロック定義のセキュリティ属性	その他の属性またはジェネリック

VHDL の組み合わせ回路

XST では、次の VHDL 組み合わせ回路がサポートされます。

- ・ [VHDL の同時処理信号代入文](#)
- ・ [VHDL のジェネレート文](#)
- ・ [VHDL の組み合わせプロセス文](#)

VHDL の同時処理信号代入文

組み合わせロジックは、同時処理信号代入文を使用して記述されます。

- ・ 同時処理信号代入文はアーキテクチャ文の本体で指定されます。
- ・ VHDL では、次の 3 種類の同時処理信号代入文があります。
 - － シンプルな信号代入文
 - － 選択文 (with-select-when)
 - － 条件文 (when-else)
- ・ 同時処理信号代入文は必要に応じていくつでも記述できます。
- ・ アーキテクチャ部分で代入した同時処理信号の順序とは関係ありません。
- ・ 同時処理信号代入はすべて同時にアクティブになります
- ・ 代入文の右側の信号の値が変化すると、同時処理代入文が再評価されます。
- ・ その再評価された結果が左側の信号に代入されます。

シンプルな信号代入文の VHDL コード例

```
T <= A and B;
```

同時選択代入文の VHDL コード例

```
--
-- Concurrent selection assignment in VHDL
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/combinatorial/concurrent_selected_assignment.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity concurrent_selected_assignment is
    generic (
        width: integer := 8);
    port (
        a, b, c, d : in std_logic_vector (width-1 downto 0);
        sel : in std_logic_vector (1 downto 0);
        T : out std_logic_vector (width-1 downto 0) );
end concurrent_selected_assignment;

architecture bhv of concurrent_selected_assignment is
begin
    with sel select
        T <= a when "00",
            b when "01",
            c when "10",
            d when others;
end bhv;
```

同時条件代入文 (when-else) の VHDL コード例

```
--
-- A concurrent conditional assignment (when-else)
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/combinatorial/concurrent_conditional_assignment.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity concurrent_conditional_assignment is
    generic (
        width: integer := 8);
    port (
        a, b, c, d : in std_logic_vector (width-1 downto 0);
        sel : in std_logic_vector (1 downto 0);
        T : out std_logic_vector (width-1 downto 0) );
end concurrent_conditional_assignment;

architecture bhv of concurrent_conditional_assignment is
begin
    T <= a when sel = "00" else
        b when sel = "01" else
        c when sel = "10" else
        d;
end bhv;
```


VHDL の generate 文

XST では、VHDL の generate 文がサポートされています。

- ・ [VHDL の for-generate 文](#)
- ・ [VHDL の if-generate 文](#)

VHDL の for-generate 文

反復構造を記述するには、for-generate 文を使用します。

for-generate 文の VHDL コード例

次の例では、for-generate 文は計算結果とこの 8 ビット加算器の各ビットのキャリーアウトを記述しています。

```
--
-- A for-generate example
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/combinatorial/for_generate.vhd
--
entity for_generate is
    port (
        A,B : in  BIT_VECTOR (0 to 7);
        CIN : in  BIT;
        SUM : out BIT_VECTOR (0 to 7);
        COUT : out BIT );
end for_generate;

architecture archi of for_generate is
    signal C : BIT_VECTOR (0 to 8);
begin
    C(0) <= CIN;
    COUT <= C(8);
    LOOP_ADD : for I in 0 to 7 generate
        SUM(I) <= A(I) xor B(I) xor C(I);
        C(I+1) <= (A(I) and B(I)) or (A(I) and C(I)) or (B(I) and C(I));
    end generate;
end archi;
```

VHDL の if-generate 文

- ・ if-generate 文は、テスト結果に基づいて、HDL ソース コードのデバイスの特定部分をアクティベートするために使用します
- ・ if-generate 文は、スタティック条件でサポートされます。

if-generate 文の例

- ・ ジェネリックはどのザイリンクス FPGA デバイス ファミリをターゲットにするか示します。
- ・ if-generate 文では、次が実行されます。
 - 特定デバイス ファミリに対してジェネリック値をテスト
 - そのデバイス ファミリ専用に記述された HDL ソース コードのセクションをアクティベート

if-generate 文内でネストされた for-generate 文の VHDL コード例

次のコード例では、ジェネリックの N ビット加算器 (ビット幅 4 ~ 32) が **if-generate** および **for-generate** 文で記述されています。

```
--
-- A for-generate nested in a if-generate
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/combinatorial/if_for_generate.vhd
--
entity if_for_generate is
  generic (
    N : INTEGER := 8);
  port (
    A,B : in BIT_VECTOR (N downto 0);
    CIN : in BIT;
    SUM : out BIT_VECTOR (N downto 0);
    COUT : out BIT );
end if_for_generate;

architecture archi of if_for_generate is
  signal C : BIT_VECTOR (N+1 downto 0);
begin
  IF_N: if (N>=4 and N<=32) generate
    C(0) <= CIN;
    COUT <= C(N+1);
    LOOP_ADD : for I in 0 to N generate
      SUM(I) <= A(I) xor B(I) xor C(I);
      C(I+1) <= (A(I) and B(I)) or (A(I) and C(I)) or (B(I) and C(I));
    end generate;
  end generate;
end archi;
```

VHDL の組み合わせプロセス

XST では、VHDL の組み合わせプロセスがサポートされます。

- ・ VHDL 組み合わせロジックは、プロセス文で記述できます。
- ・ プロセス文は、文内で代入された信号がプロセスの実行されるたびに新しい値を代入する場合、組み合わせプロセスとなります。
- ・ このような信号は、現在の値を保持しません。
- ・ プロセスには、ローカル変数を含めることができます。

メモリ エLEMENT

組み合わせプロセスから推論されるハードウェアには、メモリ エLEMENTが含まれません。

- ・ プロセス文で代入された信号すべてが、常にプロセス ブロック内のすべての可能性のあるパスで明示的に代入される場合、そのプロセスは組み合わせプロセスとなります。
- ・ if 文または case 文のすべての分岐で信号が明示的に代入されていない場合は、通常ラッチが推論されます。
- ・ XST で予測されないラッチが推論されたら、明示的に代入されていない信号の HDL コードを確認します。

センシティブティ リスト

組み合わせプロセス文には、センシティブティリストが含まれます。

- ・ センシティブティリストは、**process** キーワードの後にかっこで囲まれます。
- ・ センシティブティリストにある信号のいずれかに変化 (イベント) が起こると、プロセスが実行されます。
- ・ 組み合わせプロセスの場合、センシティブティリストには次を含める必要があります。
 - if や case などの条件で使用するすべての信号
 - 代入文の右側の信号すべて

信号がない場合

信号がセンシティブティリストにないことがあります。

- ・ センシティブティリストに信号が含まれていない場合、XST では次が実行されます。
 - 成結果が最初のデザイン仕様と異なることがあります。
 - 警告メッセージが表示されます。
 - なかった信号をセンシティブティリストに追加します。
- ・ シミュレーション中の問題を回避するには
 - HDL ソース コードに存在しない信号すべてを追加します。
 - 合成を再実行します。

VHDL の変数および信号の代入文

XST では VHDL の変数および信号代入文がサポートされます。

- ・ プロセスには、ローカル変数を含めることができます。
- ・ ローカル変数には、次の特徴があります。
 - プロセス内で宣言して使用できる
 - 通常プロセスの外側からは見えない

プロセス文内で信号代入をした VHDL コード例

```
--
-- Signal assignment in a process
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/signals_variables/signal_in_process.vhd
--
entity signal_in_process is
    port (
        A, B : in BIT;
        S : out BIT );
end signal_in_process;

architecture archi of signal_in_process is
begin
    process (A, B)
    begin
        S <= '0' ;
        if ((A and B) = '1') then
            S <= '1' ;
        end if;
    end process;
end archi;
```

プロセス文内で変数および信号代入をした VHDL コード例

```
--
-- Variable and signal assignment in a process
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/signals_variables/variable_in_process.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity variable_in_process is
  port (
    A,B      : in  std_logic_vector (3 downto 0);
    ADD_SUB  : in  std_logic;
    S        : out std_logic_vector (3 downto 0) );
end variable_in_process;

architecture archi of variable_in_process is
begin
  process (A, B, ADD_SUB)
    variable AUX : std_logic_vector (3 downto 0);
  begin
    if ADD_SUB = '1' then
      AUX := A + B ;
    else
      AUX := A - B ;
    end if;
    S <= AUX;
  end process;
end archi;
```

VHDL の if-else 文

if-else 文および if-elsif-else 文では、真偽条件 (true-false) によって実行される文が決定されます。

- ・ 条件が真と判断された場合は **if** 文が実行されます。
- ・ 条件が偽 (または **x** か **z**) と判断された場合は **else** 文が実行されます。
- ・ 複数文から成り立つブロックを if または else 分岐文内で実行できます。
- ・ **begin** と **end** キーワードが必要です。
- ・ if-else 文はネストさせることができます。

if-else 文の VHDL コード例

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux4 is
    port (
        a, b, c, d : in std_logic_vector (7 downto 0);
        sel1, sel2 : in std_logic;
        outmux : out std_logic_vector (7 downto 0));
end mux4;

architecture behavior of mux4 is
begin
    process (a, b, c, d, sel1, sel2)
    begin
        if (sel1 = '1') then
            if (sel2 = '1') then
                outmux <= a;
            else
                outmux <= b;
            end if;
        else
            if (sel2 = '1') then
                outmux <= c;
            else
                outmux <= d;
            end if;
        end if;
    end process;
end behavior;
```

VHDL の case 文

VHDL の case 文では、次が実行されます。

- ・ 論理式を比較し、複数ある並列分岐の 1 つを実行します。
- ・ 記述された順に分岐を評価します。
- ・ 最初に **true** になった分岐から実行されます。
- ・ どのブランチも一致しなかった場合は、デフォルトの分岐が実行されます。

case 文のコード例

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux4 is
  port (
    a, b, c, d : in std_logic_vector (7 downto 0);
    sel : in std_logic_vector (1 downto 0);
    outmux : out std_logic_vector (7 downto 0));
end mux4;

architecture behavior of mux4 is
begin
  process (a, b, c, d, sel)
  begin
    case sel is
      when "00" => outmux <= a;
      when "01" => outmux <= b;
      when "10" => outmux <= c;
      when others => outmux <= d; -- case statement must be complete
    end case;
  end process;
end behavior;
```

VHDL の for-loop 文

XST では、VHDL の for-loop 文がサポートされています。

- ・ 定数の範囲
- ・ 次の演算子を使用したテスト条件の停止
 - <
 - <=
 - >
 - >=
- ・ 次のいずれかに適合する次ステップの計算
 - $var = var + step$
 - $var = var - step$
 - ◆ var はループ変数
 - ◆ $step$ は定数値
- ・ **next** 文および **exit** 文

for-loop 文の VHDL コード例

```
--
-- For-loop example
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/combinatorial/for_loop.vhd
--
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity countzeros is
    port (
        a : in std_logic_vector (7 downto 0);
        Count : out std_logic_vector (2 downto 0) );
end countzeros;

architecture behavior of countzeros is
    signal Count_Aux: std_logic_vector (2 downto 0);
begin
    process (a, Count_Aux)
    begin
        Count_Aux <= "000";
        for i in a'range loop
            if (a(i) = '0') then
                Count_Aux <= Count_Aux + 1;
            end if;
        end loop;
        Count <= Count_Aux;
    end process;
end behavior;
```

VHDL の順序ロジック

VHDL の順序ロジックには、次が含まれます。

- ・ [VHDL のセンシティビティリスト付き順次プロセス文](#)
- ・ [VHDL のセンシティビティリストのない順次プロセス文](#)
- ・ [VHDL の初期値と動作に合わせたセット/リセット](#)
- ・ [VHDL のメモリー エLEMENTのデフォルト初期値](#)

VHDL のセンシティビティ リスト付き順次プロセス文

プロセスのある条件に対して代入されていない信号がある場合、VHDL プロセスは組み合わせプロセスではなく、順次プロセスになります。

内部ステートまたはメモリー (フリップフロップまたはラッチ) が生成されます。

順次ロジックは、センシティビティリスト ベースの記述方式で記述することをお勧めします。

詳細は、[第 7 章「HDL コーディング手法」](#)を参照してください。

順序ロジックの記述

プロセスを使用したセンシティビティリスト付き順次プロセス文には、次を記述します。

- ・ 次を含むセンシティビティリスト :
 - クロック信号
 - 順次ELEMENTを非同期に制御するオプションの信号 (非同期セット/リセット)
- ・ クロック イベントを記述した if 文

非同期制御ロジックの記述

- ・ 非同期制御ロジック (非同期セット/リセット) の記述がそのクロック イベント文よりも前に終了
- ・ 同期制御ロジック (データ、オプションの同期セット/リセット、オプションのクロック イネーブル) の記述がそのクロック イベントの if 分岐文よりも前に終了

この情報は、次の表にまとめられています。

非同期制御ロジックの記述のサマリ

記述	内容	実行
非同期制御ロジック	非同期セット/リセット	クロック イベント文の前
同期ロジック	<ul style="list-style-type: none"> ・ データ ・ オプションの同期セット/リセット ・ オプションのクロック イネーブル 	クロック イベントの if 分岐

センシティビティ リスト付き順次プロセス文の構文

```
process (<sensitivity list>)
begin
    <asynchronous part>
    <clock event>
    <synchronous part>
end;
```

クロック イベント文

- ・ クロック イベント文は、次のように記述します。
 - 立ち上がりエッジ

```
If clk'event and clk = '1' then
```
 - 立ち下がりエッジ

```
If clk'event and clk = '0' then
```
- ・ VHDL'93 IEEE 標準の rising_edge および falling_edge 関数を使用すると、コードはわかりやすくなります。
 - 立ち上がりエッジ

```
If rising_edge(clk) then
```
 - 立ち下がりエッジ

```
If falling_edge(clk) then
```

信号がない場合

信号がセンシティビティ リストにないことがあります。

- ・ センシティビティ リストに信号が含まれていない場合、XST では次が実行されます。
 - 成結果が最初のデザイン仕様と異なることがあります。
 - 警告メッセージが表示されます。
 - なかった信号をセンシティビティ リストに追加します。
- ・ シミュレーション中の問題を回避するには
 - HDL ソース コードに存在しない信号すべてを追加します。
 - 合成を再実行します。

VHDL のセンシティビティ リストのない順次プロセス文

XST では、wait 文を使用した順次プロセスの記述がサポートされます。

- ・ 順次プロセスはセンシティビティ リストなしで記述します。
- ・ 同じ順次プロセスにはセンシティビティ リストと wait 文は併用できません。

使用できる wait 文は 1 文のみです。

- ・ wait 文は、冒頭に記述します。
- ・ wait 文の条件で順次ロジック クロックを記述します。

wait 文を使用した順次プロセスの VHDL コード例

```
process
begin
    wait until rising_edge(clk);
    q <= d;
end process;
```

wait 文でクロック エッジを記述したコード例

クロック イネーブルは、クロックを使用した wait 文で記述できます。

```
process
begin
    wait until rising_edge(clk) and clken = '1';
    q <= d;
end process;
```

wait 文の後にクロック エッジを記述したコード例

クロック イネーブルは、別々に記述できます。

```
process
begin
    wait until rising_edge(clk);
    if clken = '1' then
        q <= d;
    end if;
end process;
```

同期制御ロジックの記述

- ・ このコード手法を使用すると、クロック イネーブルだけでなく、同期リセットまたはセットのような同期制御ロジックを記述できます。
- ・ センシティブティリストのないプロセス文を使用した非同期制御ロジックを持つ順次エレメントは記述できません。このような機能を使用できるのは、センシティブティリスト付きのプロセス文だけです。
- ・ XST では、wait 文に基づいたラッチの記述がサポートされません。
- ・ 同期ロジックを記述するには、センシティブティリスト付きプロセスを使用した方が柔軟性があるのでお勧めします。

VHDL の初期値と動作に合わせたセット/リセット

レジスタを宣言する際に初期化できます。

初期値は次のようになります。

- ・ 定数値です。
- ・ 関数呼び出しから生成できます (例：外部データ ファイルから初期値の読み込み)。
- ・ 以前の初期値に依存できません。
- ・ レジスタに伝搬するパラメーター値にできます。

レジスタを初期化する VHDL コード例 1

次のコード例は、パワーアップ (電源投入) 値を指定しています。

- ・ 順次エレメントは回路に電源が入ると初期化される
- ・ 回路グローバル リセットが適用される

```
signal arb_onebit    : std_logic := '0';  
signal arb_priority : std_logic_vector(3 downto 0) := "1011";
```

順次エレメントの動作に合わせた初期化

- ・ 順次エレメントを動作に合わせて初期化するには、次を記述する必要があります。
 - － セット/リセット値
 - － ローカル制御ロジック
- ・ レジスタのリセット ラインが最適な値になったときにレジスタに値を代入します。
- ・ 次の例を参照してください。
- ・ 次の詳細については、「[フリップフロップとレジスタ](#)」を参照してください。
 - － 動作に合わせたセット/リセット
 - － 非同期 vs 同期セット/リセット

レジスタを初期化する VHDL コード例 2

```
process (clk, rst)  
begin  
    if rst='1' then  
        arb_onebit <= '0';  
    end if;  
end process;
```

レジスタを初期化する VHDL コード例 3

次のコードは、パワーアップの初期化と動作に合わせたリセットを混ぜた例です。

```
--
-- Register initialization
-- Specifying initial contents at circuit power-up
-- Specifying an operational set/reset
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/initial/initial_1.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity initial_1 is
    Port (
        clk, rst : in std_logic;
        din : in std_logic;
        dout : out std_logic);
end initial_1;

architecture behavioral of initial_1 is
    signal arb_onebit : std_logic := '1'; -- power-up to vcc
begin

    process (clk)
    begin
        if (rising_edge(clk)) then
            if rst='1' then -- local synchronous reset
                arb_onebit <= '0';
            else
                arb_onebit <= din;
            end if;
        end if;
    end process;

    dout <= arb_onebit;

end behavioral;
```

VHDL のメモリ エLEMENTのデフォルト初期値

すべてのメモリ エLEMENTは、既知のステートになる必要があります。

- ・ すべてのメモリ エLEMENTは、既知のステートになる必要があるため、場合によっては IEEE 規格の初期値に従わない場合があります。
- ・ 次に例を示します。
 - たとえば、前述のコード例では、arb_onebit 信号が 1 に初期化されない場合、XST では初期値としてデフォルトの 0 を割り当てます。
 - XST は IEEE 規格 (std_logic のデフォルト値は U) に従いません。

初期化

初期化は、レジスタおよび RAM コンポーネントの両方で同じになります。

- ・ XST は信号の値を初期化する際に、できるだけ IEEE VHDL 規格に従います。
- ・ 初期値が VHDL コードに含まれていない場合は、「VHDL の初期値」の表の XST 列のようなデフォルト値が使用されます。

未接続のポート

未接続の出力ポートは、デフォルトで VHDL 初期値の XST 列に示されている値になります。

- ・ 出力ポートに初期状態がある場合は、未接続の出力ポートが定義された初期状態になるように接続されます。
- ・ IEEE の VHDL 仕様では、未接続の入力ポートを使用できません。
 - 未接続の入力ポートがあると、XST でエラー メッセージが表示されます。
 - 入力ポートに open キーワードが付けられていても、エラー メッセージが表示されます。

VHDL の初期値

データ型	IEEE	XST
bit	0	0
std_logic	U	0
bit_vector (3 downto 0)	0	0
std_logic_vector (3 downto 0)	0	0
integer (unconstrained)	integer'left	integer'left
integer range 7 downto 0	integer'left = 7	integer'left = 7 (コードは 111)
integer range 0 to 7	integer'left = 0	integer'left = 0 (コードは 000)
Boolean	FALSE	FALSE (コードは 0)
enum (S0,S1,S2,S3)	type'left = S0	type'left = S0 (コードは 000)

VHDL の関数とプロシージャ

VHDL の関数 (ファンクション) およびプロシージャを宣言しておく、デザインでブロックを複数回使用する場合に有益です。

- ・ 関数およびプロシージャは、次で宣言します。
 - entity の宣言部分
 - architecture 文
 - package 文
- ・ 次を含む function または procedure 文
 - 宣言部分
 - 本体
- ・ 宣言部分では次が指定されます。
 - 入力パラメーター
 - 出力と入出力パラメーター (プロシージャのみ)
 - 出力と入出力パラメーター (プロシージャのみ)
- ・ これらのパラメーターには制約を設定する必要はありません。制約を設定しないということは、パラメーターが特定の範囲に制限されないということです。
- ・ 内容は組み合わせプロセス文に類似しています。
- ・ レゾリューション関数は、IEEE std_logic_1164 パッケージで定義されるもの以外は、サポートされません。

パッケージ内で宣言された関数の VHDL コード例

次に、ADD 関数をパッケージで宣言するコード例を示します。

- ・ ADD 関数は 1 ビット加算器です。
- ・ ADD 関数は 4 ビット加算器を作成するために 4 回呼び出されます。

```
--  
-- Declaration of a function in a package  
--  
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip  
-- File: VHDL_Language_Support/functions_procedures/function_package_1.vhd  
--  
package PKG is  
    function ADD (A,B, CIN : BIT )  
        return BIT_VECTOR;  
end PKG;  
  
package body PKG is  
    function ADD (A,B, CIN : BIT )  
        return BIT_VECTOR is  
        variable S, COUT : BIT;  
        variable RESULT : BIT_VECTOR (1 downto 0);  
    begin  
        S := A xor B xor CIN;  
        COUT := (A and B) or (A and CIN) or (B and CIN);
```



```

        RESULT := COUT & S;
        return RESULT;
    end ADD;
end PKG;

use work.PKG.all;

entity EXAMPLE is
    port (
        A,B : in BIT_VECTOR (3 downto 0);
        CIN : in BIT;
        S : out BIT_VECTOR (3 downto 0);
        COUT : out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
    signal S0, S1, S2, S3 : BIT_VECTOR (1 downto 0);
begin
    S0 <= ADD (A(0), B(0), CIN);
    S1 <= ADD (A(1), B(1), S0(1));
    S2 <= ADD (A(2), B(2), S1(1));
    S3 <= ADD (A(3), B(3), S2(1));
    S <= S3(0) & S2(0) & S1(0) & S0(0);
    COUT <= S3(1);
end ARCHI;

```

パッケージ内で宣言されたプロシージャの VHDL コード例

次は、上記と同じ例ですが、プロシージャ文を使用しています。

```

--
-- Declaration of a procedure in a package
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/functions_procedures/procedure_package_1.vhd
--
package PKG is
    procedure ADD (
        A, B, CIN : in BIT;
        C : out BIT_VECTOR (1 downto 0) );
end PKG;

package body PKG is
    procedure ADD (
        A, B, CIN : in BIT;
        C : out BIT_VECTOR (1 downto 0)
    ) is
        variable S, COUT : BIT;
    begin
        S := A xor B xor CIN;

```

```

        COUT := (A and B) or (A and CIN) or (B and CIN);
        C := COUT & S;
    end ADD;
end PKG;

use work.PKG.all;

entity EXAMPLE is
    port (
        A,B : in BIT_VECTOR (3 downto 0);
        CIN : in BIT;
        S : out BIT_VECTOR (3 downto 0);
        COUT : out BIT );
end EXAMPLE;

architecture ARCH1 of EXAMPLE is
begin
    process (A,B,CIN)
        variable S0, S1, S2, S3 : BIT_VECTOR (1 downto 0);
    begin
        ADD (A(0), B(0), CIN, S0);
        ADD (A(1), B(1), S0(1), S1);
        ADD (A(2), B(2), S1(1), S2);
        ADD (A(3), B(3), S2(1), S3);
        S <= S3(0) & S2(0) & S1(0) & S0(0);
        COUT <= S3(1);
    end process;
end ARCH1;

```

再帰関数の VHDL コード例

XST では再帰関数もサポートされます。次の例は、n! 関数を使用しています。

```

function my_func(x : integer) return integer is
begin
    if x = 1 then
        return x;
    else
        return (x*my_func(x-1));
    end if;
end function my_func;

```

VHDL のアサート文

VHDL のアサート文には、次の特徴があります。

- ・ デザインのデバッグをするのに役立ちます。
- ・ 次に対して使用される不正な値など、問題のある条件文を検出できます。
 - － ジェネリック、定数、generate 条件
 - － 呼び出された関数のパラメーター

アサート文で検出されたエラーは、問題のレベルに応じて次のいずれかとして表示されます。

- ・ 警告メッセージが表示されます。または
- ・ デザインは合成できず、エラーメッセージが表示されます。

XST では、スタティック条件のアサート文のみがサポートされます。

アサート文を使用したデザイン ルール チェック

次のコード例には、シフトレジスタを記述する SINGLE_SRL というブロックが含まれています。

- ・ このシフトレジスタのサイズは、SRL_WIDTH というジェネリック値によって決定します。
- ・ アサート文により、1 つのシフトレジスタのサイズが SRL (Shift Register LUT) のサイズを超えていないかどうかチェックされます。
- ・ 次の理由から、シフトレジスタの最大サイズには 17 ビットを超える値を使用できません。
 - SRL のサイズは 16 ビット
 - XST はスライスのフリップフロップを使用してシフトレジスタの最後のステージをインプリメント
- ・ SINGLE_SRL ブロックは、TOP というエンティティ内で 2 回インスタンス化されています。
 - 1 つ目のインスタンス化
SRL_WIDTH = 13
 - 2 つ目のインスタンス化
SRL_WIDTH = 18

デザインルール チェック用アサート文の VHDL コード例

```
--
-- Use of an assert statement for design rule checking
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/asserts/asserts_1.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity SINGLE_SRL is
    generic (SRL_WIDTH : integer := 24);
    port (
        clk : in std_logic;
        inp : in std_logic;
        outp : out std_logic);
end SINGLE_SRL;

architecture beh of SINGLE_SRL is
    signal shift_reg : std_logic_vector (SRL_WIDTH-1 downto 0);
begin
    assert SRL_WIDTH <= 17
    report "The size of Shift Register exceeds the size of a single SRL"
```

```
severity FAILURE;

process (clk)
begin
    if rising_edge(clk) then
        shift_reg <= shift_reg (SRL_WIDTH-2 downto 0) & inp;
    end if;
end process;

outp <= shift_reg(SRL_WIDTH-1);
end beh;

library ieee;
use ieee.std_logic_1164.all;

entity TOP is
    port (
        clk : in std_logic;
        inp1, inp2 : in std_logic;
        outp1, outp2 : out std_logic);
end TOP;

architecture beh of TOP is
    component SINGLE_SRL is
        generic (SRL_WIDTH : integer := 16);
        port(
            clk : in std_logic;
            inp : in std_logic;
            outp : out std_logic);
    end component;
begin
    inst1: SINGLE_SRL
        generic map (SRL_WIDTH => 13)
        port map(
            clk => clk,
            inp => inp1,
            outp => outp1 );
    inst2: SINGLE_SRL
        generic map (SRL_WIDTH => 18)
        port map(
            clk => clk,
            inp => inp2,
            outp => outp2 );
end beh;
```

アサート文を使用したデザイン ルール チェックのエラー メッセージ

HDL Elaboration

* =====

Elaborating entity <TOP> (architecture <beh>) from library <work>. Elaborating entity
<SINGLE_SRL> (architecture <beh>) with generics from library <work>. Elaborating entity
<SINGLE_SRL> (architecture <beh>) with generics from library <work>. ERROR:HDLCompiler:1242 -
"VHDL_Language_Support/asserts/asserts_1.vhd" Line 15: "The size of Shift Register exceeds the
size of a single SRL": exiting elaboration "VHDL_Language_Support/asserts/asserts_1.vhd" Line 4.
netlist SINGLE_SRL(18)(beh) remains a blackbox, due to errors in its contents

VHDL ライブラリおよびパッケージ

VHDL ライブラリおよびパッケージには、次が含まれます。

- ・ [VHDL ライブラリ](#)
- ・ [VHDL の定義済みパッケージ](#)

VHDL のライブラリ

ライブラリは、デザイン ユニットをコンパイルするディレクトリです。

- ・ デザイン ユニットは、エンティティまたはアーキテクチャとパッケージになります。
- ・ VHDL および Verilog ソース ファイルはそれぞれ指定ライブラリにコンパイルされます。
- ・ 次の詳細は、「[HDL 合成プロジェクトの作成](#)」を参照してください。
 - － HDL 合成プロジェクト ファイルの構文
 - － HDL ソース ファイルがコンパイルされるライブラリの指定方法
- ・ ライブラリにコンパイルされるデザイン ユニットは、ライブラリ節を使用してそのライブラリを参照していれば、どの VHDL ソース ファイルからでも起動できます。

```
library library_name;
```

- ・ work ライブラリ :
 - － デフォルトライブラリです。
 - － ライブラリ節を必要としません。
- ・ デフォルトのライブラリ名を変更するには、次を使用します。

```
run -work_lib
```

- ・ デフォルトライブラリの物理ディレクトリ、その他ユーザー定義のライブラリの物理ディレクトリは、[work ディレクトリ](#)制約で定義されるディレクトリの下にある同じ名前のサブディレクトリです。

VHDL の定義済みパッケージ

XST では次の VHDL の定義済みパッケージがサポートされます。

- ・ [VHDL の定義済み標準パッケージ](#)
- ・ [VHDL の定義済み IEEE パッケージ](#)

VHDL の定義済みパッケージ :

- ・ **std** および **ieee standard** ライブラリで定義されます。
- ・ 既にコンパイル済みです。
- ・ ユーザーがコンパイルする必要はありません。
- ・ HDL ソースコードに直接含めることができます。

VHDL の定義済み標準パッケージ

VHDL の定義済み標準パッケージ :

- ・ デフォルトで提供されます。
- ・ 次の基本的な VHDL データ型を定義します。
 - bit
 - bit_vector
 - integer
 - natural
 - real
 - boolean

VHDL の定義済み IEEE パッケージ

XST では VHDL の定義済みの IEEE パッケージが一部サポートされます。

VHDL の定義済み IEEE パッケージ

- ・ IEEE ライブラリでコンパイル済み
- ・ 共通のデータ型、関数、プロシージャを定義

XST では、次の IEEE パッケージがサポートされます。

- ・ numeric_bit
 - bit に基づいて unsigned および signed ベクタ型
 - オーバーロードされた数値演算子、変換関数やこれらの型の拡張関数
- ・ std_logic_1164
 - std_logic、std_ulogic、std_logic_vector、std_ulogic_vector 型
 - これらのデータ型に基づいた変換関数
- ・ std_logic_arith (Synopsys)
 - std_logic に基づいて unsigned および signed ベクタ型
 - オーバーロードされた数値演算子、変換関数やこれらの型の拡張関数
- ・ numeric_std
 - std_logic に基づいて unsigned および signed ベクタ型
 - オーバーロードされた数値演算子、変換関数やこれらの型の拡張関数。これは std_logic_arith と同じです。
- ・ std_logic_unsigned (Synopsys)
std_logic および std_logic_vector の符号なし数値演算子
- ・ std_logic_signed (Synopsys)
std_logic および std_logic_vector の符号付き数値演算子
- ・ std_logic_misc (Synopsys)
and_reduce および or_reduce のような std_logic_1164 パッケージの補足タイプ、サブタイプ、定数、関数

独自の VHDL パッケージの定義

次を指定する独自の VHDL パッケージを定義できます。

- ・ タイプおよびサブタイプ
- ・ 定数
- ・ 関数とプロシージャ
- ・ コンポーネント宣言

VHDL のパッケージを定義すると、ほかのパーツのプロジェクトから共通の定義およびモデルを使用できます。

VHDL パッケージを定義するには、次が必要です。

- ・ パッケージ宣言
上記の各エレメントを宣言します。
- ・ パッケージ本体
パッケージ宣言文で宣言された関数およびプロシージャを記述します。

パッケージ宣言の構文

```
package mypackage is

    type mytype is
        record
            first : integer;
            second : integer;
        end record;

    constant myzero : mytype := (first => 0, second => 0);

    function getfirst (x : mytype) return integer;

end mypackage;
```

パッケージ本体の構文

```
package body mypackage is

    function getfirst (x : mytype) return integer is
    begin
        return x.first;
    end function;

end mypackage;
```


VHDL パッケージのアクセス

VHDL パッケージにアクセスするには、次を実行する必要があります。

1. パッケージをコンパイルするライブラリを `library` 節で含めます。

```
library library_name;
```

2. パッケージまたはパッケージに含まれる特有の定義を `use` 節で指定します。

```
use library_name.package_name.all;
```

3. これらの行は、パッケージ定義を使用するエンティティまたはアーキテクチャ文の直前に挿入する必要があります。

デフォルトのライブラリは `work` なので、指定したパッケージがこのライブラリでコンパイルされる場合は、`library` 節は必要ありません。

VHDL のファイル タイプ サポート

関 数	パッケージ
file (text タイプのみ)	standard
access (line タイプのみ)	standard
file_open (file, name, open_kind)	standard
file_close (file)	standard
endfile (file)	standard
text	std.textio
line	std.textio
width	std.textio
readline (text, line)	std.textio
readline (line, bit, boolean)	std.textio
read (line, bit)	std.textio
readline (line, bit_vector, boolean)	std.textio
read (line, bit_vector)	std.textio
read (line, boolean, boolean)	std.textio
read (line, boolean)	std.textio
read (line, character, boolean)	std.textio
read (line, character)	std.textio
read (line, string, boolean)	std.textio
read (line, string)	std.textio
write (file, line)	std.textio
write (line, bit, boolean)	std.textio
write (line, bit)	std.textio
write (line, bit_vector, boolean)	std.textio
write (line, bit_vector)	std.textio
write (line, boolean, boolean)	std.textio
write (line, boolean)	std.textio
write (line, character, boolean)	std.textio
write (line, character)	std.textio
write (line, integer, boolean)	std.textio
write (line, integer)	std.textio
write (line, string, boolean)	std.textio
write (line, string)	std.textio
read (line, std_ulogic, boolean)	ieee.std_logic_textio
read (line, std_ulogic)	ieee.std_logic_textio
read (line, std_ulogic_vector), boolean	ieee.std_logic_textio

関数	パッケージ
read (line, std_ulogic_vector)	ieee.std_logic_textio
read (line, std_logic_vector, boolean)	ieee.std_logic_textio
read (line, std_logic_vector)	ieee.std_logic_textio
write (line, std_ulogic, boolean)	ieee.std_logic_textio
write (line, std_ulogic)	ieee.std_logic_textio
write (line, std_ulogic_vector, boolean)	ieee.std_logic_textio
write (line, std_ulogic_vector)	ieee.std_logic_textio
write (line, std_logic_vector, boolean)	ieee.std_logic_textio
write (line, std_logic_vector)	ieee.std_logic_textio
hread	ieee.std_logic_textio

VHDL ファイルの読み出し/書き込み機能

XST では、制限付きで VHDL のファイル読み出しおよび書き込みがサポートされています。

ファイルの読み出し

ファイル読み込み機能を使用すると、外部データファイルからメモリを初期化できます。詳細は、「[外部データファイルでの初期内容の指定](#)」を参照してください。

ファイルの書き込み

ファイルの書き込み機能は、次の目的に使用します。

- ・ デバッグ
- ・ 特定の定数またはジェネリック値の外部ファイルへの書き込み

必須パッケージ

次のパッケージは必ず必要です。

- ・ std_logic_textio パッケージ：
 - std ライブラリから使用できます。
 - 基本的なテキスト ベースのファイル I/O 機能を提供します。
 - 次のファイル I/O 操作のプロシージャを定義します。
 - ◆ readline
 - ◆ read
 - ◆ writeline
 - ◆ write
- ・ ieee.std_logic_textio パッケージ：
 - IEEE ライブラリから使用できます。
 - その他のデータ型の拡張テキスト I/O サポートを提供します。
 - 「[VHDL のファイル タイプ サポート](#)」に示すように、read および write プロシージャをオーバーロードします。

ファイルを暗黙的/明示的に開いて閉じる方法

XST では、ファイルを開く際および閉じる際に、ファイルを暗黙的に指定する方法と、明示的に指定する方法がどちらもサポートされます。

次のように宣言すると、暗黙的に指定されたファイルが開きます。

```
file myfile : text open write_mode is "myfilename.dat"; --
declaration and implicit open
```

外部ファイルを明示的に指定して開いて閉じるには、次のように記述します。

```
file myfile : text; -- declaration

variable file_status : file_open_status;

...

file_open (file_status, myfile, "myfilename.dat", write_mode); --
explicit open

...

file_close(myfile); -- explicit close
```

外部ファイルからのメモリー内容の読み込み

詳細は、「[外部データファイルでの RAM の初期内容の指定](#)」を参照してください。

デバッグ用ファイルへの書き込み

アップデート情報は、「[はじめに](#)」のコード例を参照してください。

ファイルへの書き込み VHDL コード例 (明示的に開いて閉じる方法)

ファイルの書き込み機能は、デバッグによく使用されます。次のコード例では、書き込み関数が明示的に開かれたファイルに対して実行されています。

```
--
-- Writing to a file
-- Explicit open/close with the VHDL'93 FILE_OPEN and FILE_CLOSE procedures
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/file_type_support/filewrite_explicitopen.vhd
--

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_TEXTIO.all;
use STD.TEXTIO.all;

entity filewrite_explicitopen is
  generic (data_width: integer:= 4);
  port ( clk : in std_logic;
        di  : in std_logic_vector (data_width - 1 downto 0);
        do  : out std_logic_vector (data_width - 1 downto 0));
```

```
end fwrite_explicitopen;

architecture behavioral of fwrite_explicitopen is
    file results : text;
    constant base_const: std_logic_vector(data_width - 1 downto 0) := conv_std_logic_vector(3,data_width);
    constant new_const:  std_logic_vector(data_width - 1 downto 0) := base_const + "0100";
begin

    process(clk)
        variable txtline : line;
        variable file_status : file_open_status;
    begin
        file_open (file_status, results, "explicit.dat", write_mode);
        write(txtline,string'("-----"));
        writeline(results, txtline);
        write(txtline,string'("Base Const: "));
        write(txtline, base_const);
        writeline(results, txtline);
        write(txtline,string'("New  Const: "));
        write(txtline,new_const);
        writeline(results, txtline);
        write(txtline,string'("-----"));
        writeline(results, txtline);
        file_close(results);
    if rising_edge(clk) then
        do <= di + new_const;
    end if;
    end process;

end behavioral;
```

ファイルへの書き込み VHDL コード例 (暗示的に開いて閉じる方法)

次はファイルを暗黙的に指定して開いた場合の例です。

```
--
-- Writing to a file. Implicit open/close
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/file_type_support/filewrite_implicitopen.vhd
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_TEXTIO.all;
use STD.TEXTIO.all;

entity filewrite_implicitopen is
    generic (data_width: integer:= 4);
    port ( clk : in  std_logic;
          di  : in  std_logic_vector (data_width - 1 downto 0);
          do  : out std_logic_vector (data_width - 1 downto 0));
end filewrite_implicitopen;

architecture behavioral of filewrite_implicitopen is
    file results : text open write_mode is "implicit.dat";
    constant base_const: std_logic_vector(data_width - 1 downto 0) := conv_std_logic_vector(3,data_width);
    constant new_const:  std_logic_vector(data_width - 1 downto 0) := base_const + "0100";
begin

    process(clk)
        variable txtline : LINE;
    begin
        write(txtline,string'("-----"));
        writeline(results, txtline);
        write(txtline,string'("Base Const: "));
        write(txtline,base_const);
        writeline(results, txtline);
        write(txtline,string'("New  Const: "));
        write(txtline,new_const);
        writeline(results, txtline);
        write(txtline,string'("-----"));
        writeline(results, txtline);

        if rising_edge(clk) then
            do <= di + new_const;
        end if;
    end process;

end behavioral;
```

書き込み関数を使用したデバッグ

書き込み関数を使用してデバッグするには、次のような規則に従う必要があります。

- ・ std_logic で read 操作中には、次に注意してください。
 - 使用できる文字は、0、1、およびスペースのみです。
 - X や Z のようなそれ以外の値は使用できません。
 - ファイルに使用できない文字が含まれる場合、そのデザインは拒否されます。
- ・ 別のディレクトリでも同じファイル名は使用しないでください。
- ・ read プロシージャへの条件呼び出しは使用しないでください。

```
if SEL = '1' then
    read (MY_LINE, A(3 downto 0));
else
    read (MY_LINE, A(1 downto 0));
end if;
```

VHDL 構文

VHDL の構文には、次があります。

- ・ [VHDL のデザイン エンティティとコンフィギュレーション](#)
- ・ [VHDL の論理式](#)
- ・ [VHDL 文](#)

VHDL のデザイン エンティティとコンフィギュレーション

XST では、次以外の VHDL デザイン エンティティとコンフィギュレーションがサポートされます。

VHDL エンティティ ヘッダー

- ・ ジェネリック
サポートあり
- ・ ポート
サポートあり (制約なしのポートを含む)
- ・ エンティティ文
サポートなし

VHDL パッケージ

- ・ STANDARD
- ・ TIME はサポートなし

VHDL の物理型

- ・ TIME
無視
- ・ REAL
サポートあり (定数計算関数でのみ)

VHDL モード

- ・ リンケージ
- ・ サポートなし

VHDL の宣言

- ・ データ型
- ・ 次がサポートされます。
 - 列挙型
 - 定数範囲の正の値の型
 - ビット ベクタ型
 - 多次元配列

VHDL のオブジェクト

- ・ 定数宣言
サポートあり (ディファード定数を除く)
- ・ 信号宣言
サポートあり (レジスタまたはバス タイプの信号を除く)
- ・ 属性宣言
一部の属性のみサポートし、ほかは無視。

詳細は、第 9 章「[デザイン制約](#)」を参照してください。

VHDL の詳細設定

- ・ 次のような一部の定義済み属性のみをサポート
 - HIGHLOW
 - LEFT
 - RIGHT
 - RANGE
 - REVERSE_RANGE
 - LENGTH
 - POS
 - ASCENDING
 - EVENT
 - LAST_VALUE
- ・ コンフィギュレーション
インスタンスリストの all 節のみでサポート。節がない場合、デフォルトライブラリにコンパイルされているエンティティ/アーキテクチャを使用。
- ・ 接続解除
サポートなし
- ・ オブジェクト名には、DATA_1 のように通常アンダースコア (_) を含めることができますが、XST では _DATA_1 のように信号名の冒頭文字にアンダースコアを使用できません。

VHDL の論理式

XST では、次の VHDL 論理式がサポートされています。

- ・ VHDL 演算子のサポート
- ・ VHDL オペランドのサポート

VHDL 演算子のサポート

演算子	ステータス
論理演算子：and、or、nand、nor、xor、xnor、not	サポートあり
比較演算子：=, /=, <, <=, >, >=	サポートあり
& (連結)	サポートあり
加算/減算演算子：+, -	サポートあり
*	サポートあり
/	右のオペランドが 2 のべき乗の定数の場合またあ両方のオペランドが定数の場合にサポート
rem	右のオペランドが 2 のべき乗の定数の場合のみサポート
mod	右のオペランドが 2 のべき乗の定数の場合のみサポート
シフト演算子：sll、srl、sla、sra、rol、ror	サポートあり
abs	サポートあり
**	左のオペランドが 2 の場合のみサポート
符号演算子：+, -	サポートあり

VHDL オペランドのサポート

オペランド	ステータス
抽象リテラル	整数リテラルのみサポート
物理リテラル	無視
列挙リテラル	サポートあり
文字列リテラル	サポートあり
ビット文字列リテラル	サポートあり
レコード集合	サポートあり
配列集合	サポートあり
関数呼び出し	サポートあり
条件付き論理式	定義済み属性でサポート
型変換	サポートあり
アロケータ	サポートなし
スタティックな論理式	サポートあり

VHDL 文

XST では、次の VHDL 文がサポートされています。

- ・ [VHDL の wait 文](#)
- ・ [VHDL のループ文](#)
- ・ [VHDL の同時処理文](#)

VHDL の wait 文

wait 文	ステータス
<ul style="list-style-type: none"> ・ boolean_expression まで sensitivity_list を待機状態にします。 ・ 「VHDL の組み合わせ回路」を参照してください。 	<ul style="list-style-type: none"> ・ センシティブティリストとブール代数式内の 1 つの信号でサポート。 ・ 複数の wait 文はサポートなし。 ・ ラッチの記述に wait 文はサポートされていません。
<ul style="list-style-type: none"> ・ time_expression を待ちます。 ・ 「VHDL の組み合わせ回路」を参照してください。 	サポートなし
アサート文	スタティック条件のみサポート
信号代入文	<ul style="list-style-type: none"> ・ サポートあり ・ 遅延は無視
変数代入文	サポートあり
プロシージャ呼び出し文	サポートあり
if 文	サポートあり
case 文	サポートあり

VHDL のループ文

ループ文	ステータス
for... loop... end loop	<ul style="list-style-type: none"> ・ 定数範囲のみサポート。 ・ disable 文はサポートされていません。
while... loop... end loop	サポートあり
loop ... end loop	複数の wait 文のみサポート
next 文	サポートあり
exit 文	サポートあり
return 文	サポートあり
null 文	サポートあり

VHDL の同時処理文

同時処理文	ステータス
プロセス文	サポートあり
同時処理プロシージャ呼び出し文	サポートあり
同時処理アサート文	無視
同時信号代入文	<ul style="list-style-type: none"> ・ サポートあり ・ after 節、transport/guarded オプション、波形を除く ・ UNAFFECTED はサポートあり
コンポーネント インスタンス文	サポートあり
for-generate	定数範囲のみサポート
if-generate	スタティック条件のみサポート

VHDL の予約語

abs	access	after	alias
all	and	architecture	array
assert	attribute	begin	block
body	buffer	bus	case
component	configuration	constant	disconnect
downto	else	elsif	end
entity	exit	file	for
function	generate	generic	group
guarded	if	impure	in
inertial	inout	is	label
library	linkage	literal	loop
map	mod	nand	new
next	nor	not	null
of	on	open	or
others	out	package	port
postponed	procedure	process	pure
range	record	register	reject
rem	report	return	rol
ror	select	severity	signal
shared	sla	sll	sra
srl	subtype	then	to
transport	type	unaffected	units
until	use	variable	wait
when	while	with	xnor
xor			

Verilog サポート

XST では、特に記述のない限り Verilog がサポートされます。

Verilog デザイン

複雑な回路は通常、トップ ダウン手法を使用して設計します。

- ・ 設計プロセスの各段階で、さまざまなレベルでの仕様が必要となります。たとえばアーキテクチャレベルでは、仕様はブロック図または ASM (Algorithmic State Machine) チャートに対応します。
- ・ ブロックまたは ASM 段階では、次のような N ビット ワイヤで接続されるレジスタ転送ブロックに対応します。
 - レジスタ
 - 加算器
 - カウンター
 - マルチプレクサ
 - グルー ロジック
 - Finite State Machine (FSM)
- ・ Verilog では、ASM チャートや回路図などをコンピュータ言語で記述できます。

Verilog の機能

Verilog ではデザインのビヘイビア記述または構造記述が可能で、さまざまな抽象度でデザイン オブジェクトを表現できます。

- ・ Verilog を使用してハードウェアを設計すると、次のようなソフトウェアの概念を利用できます。
 - 並列処理
 - オブジェクト指向プログラム
- ・ Verilog の構文は C 言語および Pascal に類似しています。
- ・ XST では、IEEE 1364 がサポートされています。
- ・ XST でサポートされる Verilog では、グローバル回路および各ブロックを効率的に記述できます。
 - 各ブロックに最適なフローを使用して合成されます。
 - ここで合成とは、Verilog のビヘイビア記述と構造記述を、フラット化されたゲートレベルのネットリストにコンパイルすることを指します。生成されたネットリストは、Virtex® デバイスなどのプログラマブル ロジック デバイスをカスタム プログラムするために使用できます。
 - 次には、それぞれ異なる合成方法が使用されます。
 - ◆ 数値演算ブロック
 - ◆ グルー ロジック
 - ◆ FSM コンポーネント

詳細情報

- ・ 基本的な Verilog の概念については、次を参照してください。
『IEEE Verilog HDL Reference Manual』
- ・ Verilog のビヘイビア記述については、次を参照してください。
[第 5 章「Verilog ビヘイビア言語」](#)
- ・ XST での Verilog 構文とメタ コメントの詳細は、次を参照してください。
 - [第 9 章「デザイン制約」](#)
Verilog デザイン制約とオプション
 - [Verilog -2001 の属性とメタ コメント](#)
Verilog 属性の構文
 - [第 10 章「一般制約」](#)
ISE® Design Suite のプロセス ウィンドウで Verilog オプションを設定します。

Verilog-2001 のサポート

XST では、次の Verilog 2001 の機能がサポートされています。

- ・ generate 文
- ・ ポートとデータ型を 1 つの文で宣言
- ・ ANSI 形式のポート リスト
- ・ モジュール パラメーター ポート リスト
- ・ ANSI C 形式のタスク/関数宣言
- ・ カンマで区切ったセンシティビティ リスト
- ・ 組み合わせロジック センシティビティ
- ・ 継続代入文のデフォルト ネット
- ・ デフォルト ネット宣言のディスエーブル
- ・ インデックスの付いたベクタの部分選択
- ・ 多次元配列
- ・ net および real データ型の配列
- ・ 配列のビットおよびビット部分選択
- ・ 符号付きレジスタ、ネット、およびポート宣言
- ・ 符号付き整数
- ・ 符号付き演算式
- ・ 算術シフト演算子
- ・ 32 ビットを超えるビットの自動的な幅拡張
- ・ べき乗演算子
- ・ N ビットのパラメーター
- ・ インライン パラメーターの明示
- ・ 固定ローカル パラメーター
- ・ 条件付きコンパイルの拡張
- ・ ファイルおよび行のコンパイラ指示子
- ・ 可変部分ビット選択
- ・ 再帰タスクおよび関数
- ・ 定数関数

詳細は、次を参照してください。

- ・ Sutherland, Stuart 著『Verilog 2001: A Guide to the New Features of the VERILOG Hardware Description Language』(2002)
- ・ 『IEEE Standard Verilog Hardware Description Language Manual』(IEEE Standard 1364-2001)

Verilog の変数による部分的ビット選択

Verilog -2001 では、変数を使用してベクタからビットのグループを選択できます。

- ・ 部分的なビットを選択する変数は、2 つの明示的な値で囲むのではなく、次によって定義されます。
 - 範囲の開始位置
 - ベクターの幅
- ・ パーツ選択の開始位置は、さまざまです。
- ・ パーツ選択の幅は、定数のままです。

変数による部分的ビット選択のためのシンボル

シンボル	説明
+ (プラス)	部分的なビットは開始位置から上方向で選択されます。
- (マイナス)	部分的なビットは開始位置から下方向に選択されます。

部分的なビット選択の Verilog コード例

```
reg [3:0] data;
reg [3:0] select; // a value from 0 to 7
wire [7:0] byte = data[select +: 8];
```


Verilog 構造記述

Verilog 構造記述 :

- ・ 複数のブロックを組み合わせます。
- ・ デザインを階層構造にできます。

ハードウェア構造の基本概念

概念	説明
コンポーネント	基本ブロックの構築
ポート	コンポーネントの I/O コネクタ
信号	コンポーネント間のワイヤに対応

Verilog コンポーネント

項目	表示	内容
宣言	外部	コンポーネント ポートも含めた外観
本体	内部	コンポーネントのビヘイビアや構造

- ・ コンポーネントはデザイン モジュールで表されます。
- ・ コンポーネント間の接続は、コンポーネント インスタンスエーション文で定義されます。
- ・ コンポーネント インスタンスエーション文には、次の特徴があります。
 - あるコンポーネントを別のコンポーネントまたは回路で使用する場合に、インスタンスを指定します。
 - 識別子で区別されます。
 - ローカル コンポーネント宣言部分で宣言されたコンポーネントの名前が指定されます。
 - 関係リスト (かっこで囲まれたリスト) が含まれます。このリストでは、該当するローカル ポートに関連する信号とポートが指定されます。

ビルトイン論理ゲート

Verilog には多数の論理ゲートが組み込まれています。

- ・ ロジック ゲートをインスタンスエートして大型の論理回路を構築します。
- ・ 含まれるビルトイン論理ゲートは、次のとおりです。
 - AND
 - OR
 - XOR
 - NAND
 - NOR
 - NOT

2 入力の XOR 関数の Verilog コード例

次のコード例では、組み込まれているモジュールの各インスタンスには、次のような固有のインスタンス名が指定されています。

```
· a_inv
· b_inv
· out

module build_xor (a, b, c);
    input a, b;
    output c;
    wire c, a_not, b_not;

    not a_inv (a_not, a);
    not b_inv (b_not, b);
    and a1 (x, a_not, b);
    and a2 (y, b_not, a);
    or out (c, x, y);
endmodule
```

半加算器の Verilog コード例

次は、4 つの 2 入力 NAND モジュールで構成される半加算器の構造記述です。

```
module halfadd (X, Y, C, S);
    input X, Y;
    output C, S;
    wire S1, S2, S3;

    nand NANDA (S3, X, Y);
    nand NANDB (S1, X, S3);
    nand NANDC (S2, S3, Y);
    nand NANDD (S, S1, S2);
    assign C = S3;
endmodule
```

定義済みプリミティブのインスタンス化

- Verilog の構造記述では、次のようなあらかじめ定義されているプリミティブをインスタンス化して、回路を記述できます。
 - ゲート
 - レジスタ
 - CLKDLL および BUFGのようなザイリンクス特有のプリミティブ
- これらのプリミティブは、次のようになります。
 - Verilog のプリミティブに追加されます。
 - XST Verilog ライブラリ (unisim_comp.v) で提供されます。

FDC と BUFG プリミティブをインスタンス化する Verilog コード例

FDC と BUFG の定義は、unisim_comp.v ライブラリ ファイルに含まれます。

```
module example (sysclk, in, reset, out);
    input sysclk, in, reset;
    output out;
    reg out;
    wire sysclk_out;

    FDC register (out, sysclk_out, reset, in); //position based referencing
    BUFG clk (.O(sysclk_out),.I(sysclk)); //name based referencing
    ...
endmodule
```

Verilog パラメーター

Verilog パラメーターの特徴 :

- 簡単に再利用および拡張可能なパラメーター指定したコードを作成できる
- コードをより可読性のある、コンパクトで維持しやすくする
- このような機能は次のように記述します。
 - ◆ バス サイズ
 - ◆ デザイン ユニットの特定の再帰的エレメントの量
- Verilog パラメーターは制約で、
パラメーター モジュールの各インスタンスエーションの場合、デフォルトのパラメーター値は上書き可能
- VHDL のジェネリックと同等

null 文字列パラメーターは、サポートされていません。

ジェネリック を使用すると、最上位レベルのブロックで定義される Verilog のパラメーターを再定義できます。ソースコードを変更しなくてもデザインを簡単に修正できます。これは、IP コアの生成やフロー テストで便利な機能です。

Verilog パラメーターのコード例

このコード例では、モジュール lpm_reg を 8 ビット幅でインスタンス化しているため、インスタンス buf_373 の幅が 8 ビットになっています。

```
//  
// A Verilog parameter allows to control the width of an instantiated  
// block describing register logic  
//  
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip  
// File: Verilog_Language_Support/parameter/parameter_1.v  
//  
module myreg (clk, clken, d, q);  
  
    parameter SIZE = 1;  
  
    input          clk, clken;  
    input  [SIZE-1:0] d;  
    output reg [SIZE-1:0] q;  
  
    always @(posedge clk)  
    begin  
        if (clken)  
            q <= d;  
    end  
  
endmodule  
  
module parameter_1 (clk, clken, di, do);  
  
    parameter SIZE = 8;  
  
    input          clk, clken;  
    input  [SIZE-1:0] di;  
    output  [SIZE-1:0] do;  
  
    myreg #8 inst_reg (clk, clken, di, do);  
  
endmodule
```

Verilog パラメーターと generate-for のコード例

次の例は、パラメーターと generate-for 構文を使用して反復エレメントの作成を制御する方法を示しています。詳細は、「[generate ループ文](#)」を参照してください。

```
//
// A shift register description that illustrates the use of parameters and
// generate-for constructs in Verilog
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: Verilog_Language_Support/parameter/parameter_generate_for_1.v
//
module parameter_generate_for_1 (clk, si, so);

    parameter SIZE = 8;

    input  clk;
    input  si;
    output so;

    reg [0:SIZE-1] s;

    assign so = s[SIZE-1];

    always @ (posedge clk)
        s[0] <= si;

    genvar i;
    generate
        for (i = 1; i < SIZE; i = i+1)
            begin : shreg
                always @ (posedge clk)
                    begin
                        s[i] <= s[i-1];
                    end
            end
    endgenerate

endmodule
```

Verilog パラメーターと属性の競合

Verilog パラメーターと属性が競合するのは、次のような場合です。

- ・ パラメーターおよび属性は Verilog コードのインスタンスとモジュールの両方に適用できる
- ・ 属性は制約ファイルでも指定できる

Verilog パラメーターと属性の優先順位

競合を回避するために、XST では次の優先順位の規則が使用されます。

- ・ インスタンス (下位レベル) での指定がモジュール (上位レベル) での指定より優先されます。
- ・ パラメーターと属性が同じインスタンスまたはコンポーネントに指定される場合、パラメーターが優先され、警告メッセージが表示されます。
- ・ XCF (XST 制約ファイル) で指定される属性は、VHDL コードで指定される属性またはパラメーターよりも優先されます。

XST でインスタンスに指定されている属性がモジュールに指定されているパラメーターを上書きする場合、シミュレーション ツールではそのパラメーターを使用はしますが、合成後の結果でシミュレーション不一致になります。

モジュール定義のセキュリティ属性は、ほかのどの属性またはパラメーターよりも優先されます。

この情報は、次の表にまとめられています。

Verilog パラメーターと属性の優先順位のサマリ

	インスタンスのパラメーター	モジュールのパラメーター
インスタンスの属性	パラメーターを適用(XST で警告メッセージが表示される)	属性を適用 (シミュレーションで不一致の可能性あり)
モジュールの属性	パラメーターを適用	パラメーターを適用(XST で警告メッセージが表示される)
XCF に含まれる属性	属性を適用(XST で警告メッセージが表示される)	属性を適用

Verilog の使用制限

Verilog には、次のような使用制限があります。

- ・ [大文字/小文字の区別](#)
- ・ [ブロックおよびノンブロッキング代入文](#)
- ・ [整数処理](#)

大文字/小文字の区別

XST では、潜在的な名前の競合とは関係なく、Verilog の大文字/小文字が区別されます。

- ・ Verilog では大文字と小文字が区別されるため、モジュール名、インスタンス名、信号名の大文字と小文字を変更するだけでそれらが異なるものとして認識されます。
 - XST では大文字/小文字だけが違うインスタンスや信号名を含むデザインが合成されます。
 - モジュール名が同じで大文字/小文字だけが違う場合はエラーになります。
- ・ オブジェクト名を固有のものにするために、大文字/小文字だけで区別するのはお勧めしません。大文字/小文字だけで区別すると、次のような問題が発生する可能性があります。
 - 混合言語プロジェクトで問題になる
 - XCF (ザイリンクス制約ファイル) を使用して制約を適用できなくなる

ブロッキングおよびノンブロッキング代入文

XST では、ブロッキングおよびノンブロッキング代入文がサポートされます。

- ・ ブロッキング代入文とノンブロッキング代入文は混合しないでください。
- ・ 混合すると、XST ではエラーなしに合成されても、シミュレーションでエラーになることがあります。

使用できないコード例 1

同じ信号に対してブロッキング代入文とノンブロッキング代入文は混合できません。

```
always @(in1)
begin
    if (in2)
        out1 = in1;
    else
        out1 <= in2;
end
```


使用できない コード例 2

同じ信号の別のビットに対してブロッキング代入文とノンブロッキング代入文は混合できません。

```
if (in2)
begin
    out1[0] = 1'b0;
    out1[1] <= in1;
end
else
begin
    out1[0] = in2;
    out1[1] <= 1'b1;
end
```

整数処理

XST では、整数がほかの合成ツールと異なる方法で処理される場合があるので、コードを記述する際に注意が必要です。

case 文での整数処理

case 文でビット サイズが指定されていない整数が使用されると、結果が予測不可能になります。

case 文での整数処理のコード例

- ・ 次の例では、case 文の最初に使用される 4 のビットが指定されていないので、結果が予測不可能になっています。
- ・ この問題を回避するには case 文の 4 を 3 ビットに指定します。

```
reg [2:0] condition1;
always @(condition1)
begin
  case(condition1)
    4 : data_out = 2; // < will generate bad logic
    3'd4 : data_out = 2; // < will work
  endcase
end
```

連結文での整数処理

Verilog 連結文でビット指定のない整数を使用すると、結果が予測不可能になります。

ビット サイズが指定されない式を使用すると、次のようになります。

- ・ その式が一時的な信号に代入されます。
- ・ 連結文の一時信号を次のように使用します。

```
reg [31:0] temp;
assign temp = 4'b1111 % 2;
assign dout = {12/3,temp,din};
```

Verilog -2001 の属性とメタ コメント

Verilog -2001 の属性とメタ コメントには、次が含まれます。

- ・ [Verilog-2001 の属性](#)
- ・ [Verilog のメタ コメント](#)

Verilog-2001 の属性

- ・ Verilog-2001 属性は、合成ツールなどのプログラムに特定の情報を渡すために使用します。
- ・ Verilog-2001 の属性は広く使用されています。
- ・ Verilog-2001 の属性は、モジュール宣言およびインスタネーション内で、演算子または信号に指定できます。
- ・ コンパイラでその他の属性宣言がサポートされていても、XST では無視されます。
- ・ Verilog-2001 属性は、次の場合に使用できます。
 - 次のような個々のオブジェクトに制約を設定する場合
 - ◆ モジュール
 - ◆ インスタンス
 - ◆ ネット
 - 次の合成制約を設定する場合
 - ◆ [フル ケース](#)
 - ◆ [パラレル ケース](#)

Verilog のメタ コメント

- ・ Verilog メタ コメントは、Verilog 解析ツールで認識されます。
- ・ Verilog メタ コメントは、次のような個々のオブジェクトに制約を設定します。
 - モジュール
 - インスタンス
 - ネット
- ・ Verilog メタ コメントは、合成で次の制約を設定します。
 - parallel_case および full_case
 - translate_on および translate_off
 - syn_sharing などツール専用の制約すべて

詳細は、[第 9 章「デザイン制約」](#)を参照してください。

Verilog のメタ コメントのサポート

XST では、次がサポートされます。

- ・ C 言語スタイルおよび Verilog スタイルのメタ コメント

- C スタイル

```
/* ...*/
```

C 言語スタイルでは、コメントを複数行にできます。

- Verilog スタイル

```
// ...
```

Verilog コメント スタイルは、行の末尾に追加します。

- ・ [Translate Off](#) と [Translate On](#)

```
// synthesis translate_on  
// synthesis translate_off
```

- ・ [パラレル ケース](#)

```
// synthesis parallel_case full_case  
// synthesis parallel_case  
// synthesis full_case
```

- ・ 各オブジェクトに対する制約

Verilog メタ コメント構文

```
// synthesis attribute [of] ObjectName [is] AttributeValue
```

Verilog メタ コメントのコード例

```
// synthesis attribute RLOC of u123 is R11C1.S0  
// synthesis attribute HUSER u1 MY_SET  
// synthesis attribute fsm_extract of State2 is "yes"  
// synthesis attribute fsm_encoding of State2 is "gray"
```

Verilog 構文

VHDL の構文には、次があります。

- ・ Verilog の定数
- ・ Verilog のデータ型
- ・ Verilog の継続代入文
- ・ Verilog の手続き代入文
- ・ Verilog のデザイン階層
- ・ Verilog コンパイラ指示子

Verilog の定数

定数	ステータス
整数	サポートあり
実数	サポートあり
文字列	サポートなし

Verilog のデータ型

データ型	カテゴリ	ステータス
ネット タイプ	<ul style="list-style-type: none"> ・ tri0 ・ tri1 ・ trireg 	サポートなし
駆動電流	すべて	無視
レジスタ	real および realtime レジスタ	サポートなし
名前付きイベント	すべて	サポートなし

Verilog の継続代入文

継続代入文	ステータス
駆動電流	無視
遅延	無視

Verilog の手続き代入文

手続き代入文	ステータス
assign	制限付きでサポートあり。詳細は、「assign 文および deassign 文」を参照してください。
deassign	制限付きでサポートあり。詳細は、「assign 文および deassign 文」を参照してください。
force	サポートなし
release	サポートなし
forever 文	サポートなし
repeat 文	サポートあり (repeat 値は定数にする必要あり)
for 文	サポートあり (範囲はスタティックな値にする必要あり)
delay (#)	無視
event (@)	サポートなし
wait	サポートなし
指定イベント	サポートなし
パラレル ブロック	サポートなし
指定ブロック	無視
ディスエーブル	for および repeat ループ文以外はサポートあり

Verilog のデザイン階層

デザイン階層	ステータス
モジュール定義	サポートあり
マクロ モジュール定義	サポートなし
階層名	サポートなし
defparam	サポートあり
インスタンスの配列	サポートあり

Verilog コンパイラ指示子

コンパイラ指示子	ステータス
'celldefine 'endcelldefine	無視
'default_nettype	サポートあり
'define	サポートあり
'ifdef 'else 'endif	サポートあり
'undef, 'ifndef, 'elsif,	サポートあり
'include	サポートあり
'resetall	無視

コンパイラ指示子	ステータス
'timescale	無視
'unconnected_drive 'nounconnected_drive	無視
'uselib	サポートなし
'file, 'line	サポートあり

Verilog のシステム タスクおよび関数

XST では次の表に示すシステム タスクまたは関数がサポートされます。サポートされないシステム タスクは無視されます。

システム タスクと関数	ステータス	コメント
\$display	サポートあり	エスケープ シーケンスは %d、%b、%h、%o、%c、および %s に制限されます。
\$fclose	サポートあり	
\$fdisplay	サポートあり	
\$fgets	サポートあり	
\$finish	サポートあり	\$finish はアクティブになることのない条件分岐文でのみサポートされます。
\$fopen	サポートあり	
\$fscanf	サポートあり	エスケープ シーケンスは %b および %d に制限されます。
\$fwrite	サポートあり	
\$monitor	無視	
\$random	無視	
\$readmemb	サポートあり	
\$readmemh	サポートあり	
\$signed	サポートあり	
\$stop	無視	
\$strobe	無視	
\$time	無視	
\$unsigned	サポートあり	
\$write	サポートあり	エスケープ シーケンスは %d、%b、%h、%o、%c、および %s に制限されます。
その他すべて	無視	

変換関数の使用

\$signed および \$unsigned システム タスクは、どの式でも次の構文を使用して呼び出すことができます。

`$signed(expr)` または `$unsigned(expr)`

- ・ これらの呼び出しの戻り値は入力値と同じサイズです。
- ・ 以前の符号にかかわらず、システム タスクで指定した符号に強制されます。

ファイル I/O タスクを使用したメモリ 内容の読み込み

\$readmemb および \$readmemh システム タスクは、ブロック メモリ の初期化に使用できます。

- ・ 2 進数の場合は \$readmemb を使用します。
- ・ 16 進数の場合は \$readmemh を使用します。
- ・ XST とシミュレータで処理の違いが発生しないようにするため、次のようにインデックス パラメーターを使用します。

```
$readmemb("rams_20c.data", ram, 0, 7);
```

- ・ 詳細は、「[外部データファイルでの RAM の初期内容の指定](#)」を参照してください。

ディスプレイ タスク

ディスプレイ タスクは、次のために使用します。

- ・ 情報をコンソールに表示
- ・ 外部ファイルに書き出し

これらのタスクは、initial ブロックから呼び出す必要があります。

サポートされるエスケープ シーケンス

- ・ %h
- ・ %d
- ・ %o
- ・ %b
- ・ %c
- ・ %s

Verilog の構文例

次は、2 進数の定数値を 10 進数でレポートする構文です。

```
parameter c = 8'b00101010;  
  
initial  
begin  
    $display ("The value of c is %d", c);  
end
```

Verilog ログ ファイルの例

HDL 解析段階で、次の情報がログ ファイルに記述されます。

```
Analyzing top module <example>.  
c = 8'b00101010  
"foo.v" line 9: $display : The value of c is 42
```

\$finish を使用したデザイン ルール チェックの作成

XST では、シミュレーション コントロール タスクの \$finish が部分的にサポートされます。

- ・ \$finish を使用すると、ビルトインのデザイン ルール チェック (DRC) を作成できます。
- ・ デザイン ルール チェックでは、次のデザイン コンフィギュレーションが検出されます。
 - 正しい構文で記述されている
 - 機能しなかったり、希望通りのインプリメンテーションにならないこともある
- ・ \$finish を使用すると、問題がある状態が検出されると XST を早期に停止させることができますので、合成およびインプリメンテーションにかかる時間が大幅に節約できます。
- ・ 実行状態がシミュレーションまたはボードの回路動作中に特定のダイナミックな状況に依存する場合、XST は \$finish を無視します。
 - このような状況を検出できるのは、シミュレーション ツールのみ
 - XST も含めた合成ツールではこれらが無視される

\$finish の使用を無視する Verilog コード例

```
//  
// Ignored use of $finish for simulation purposes only  
//  
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip  
// File: Verilog_Language_Support/system_tasks/finish_ignored_1.v  
//  
module finish_ignored_1 (clk, di, do);  
  
    input        clk;  
    input        [3:0] di;  
    output reg [3:0] do;  
  
    initial  
    begin  
        do = 4'b0;  
    end  
  
    always @(posedge clk)  
    begin  
        if (di < 4'b1100)  
            do <= di;  
        else  
            begin  
                $display("%t, di value %d should not be more than 11", $time, di);  
                $finish;  
            end  
        end  
    end  
  
endmodule
```

XST での \$finish のサポート

ダイナミックな状況では、\$finish システム タスクがフラグされてから無視されます。

- ・ \$finish は、実行状態が Verilog ソース コードのエラボレーション中に完全に評価できるようなスタティックな状況でのみ XST で認識されます。
 - スタティックに評価される状況の場合、主にパラメーターと予測値が比較されます。
 - この比較は、通常次に示すようなモジュール初期ブロックで実行されます。
- ・ \$display システム タスクと \$finish を併用すると、早期に停止された場合の主原因を示すメッセージが作成されます。
- ・ XST は、Verilog シミュレーション コントロール タスクの \$stop を無視します。

デザインルール チェックのために \$finish を使用する Verilog コード例

```
//  
// Supported use of $finish for design rule checking  
//  
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip  
// File: Verilog_Language_Support/system_tasks/finish_supported_1.v  
//  
module finish_supported_1 (clk, di, do);  
  
    parameter integer WIDTH    = 4;  
    parameter      DEVICE     = "virtex6";  
  
    input          clk;  
    input [WIDTH-1:0] di;  
    output reg [WIDTH-1:0] do;  
  
    initial  
    begin  
        if (DEVICE != "virtex6")  
        begin  
            $display ("DRC ERROR: Unsupported device family: %s.", DEVICE);  
            $finish;  
        end  
        if (WIDTH < 8)  
        begin  
            $display ("DRC ERROR: This module not tested for data width: %d. Minimum allowed width is 8.", WIDTH);  
            $finish;  
        end  
    end  
  
    always @(posedge clk)  
    begin  
        do <= di;  
    end  
  
endmodule
```

Verilog プリミティブ

XST では次の表に示していない限り、すべての Verilog ゲートレベルのプリミティブがサポートされます。

XST では次のような Verilog のスイッチ レベルのプリミティブはサポートされません。

- ・ cmos、nmos、pmos、rcmos、rnmos、rpmos
- ・ rtran、rtranif0、rtranif1、tran、tranif0、tranif1

XST でサポートされない Verilog のゲート レベルのプリミティブ

プリミティブ	ステータス
プルダウンとプルアップ	サポートなし
駆動電力と遅延	無視
プリミティブの配列	サポートなし

ゲート レベルのプリミティブ構文

```
gate_type instance_name (output, inputs,...);
```

ゲート レベルのプリミティブのコード例

```
and U1 (out, in1, in2); bufif1 U2 (triout, data, trienable);
```

Verilog ユーザー定義プリミティブ (UDP)

UDP を使用すると、ステート テーブル形式の機能を記述できます。

- ・ ステート テーブルには、次が含まれます。
 - すべての入力値の組み合わせを列挙
 - 回路独自の出力に対応する値を指定
- ・ UDP を使用して記述する機能は、次のいずれかにできます。
 - 組み合わせ
 - 順次
- ・ UDP は、次のような複雑ではない機能を記述するのに便利です。
 - 単純な組み合わせファンクション
 - 基本的な順次エレメント
- ・ 回路記述の詳細には、次の方法をお勧めします。
 - Verilog ビヘイビア記述を使用
 - XST の推論機能を使用
- ・ Verilog と VHDL の推論機能とコード ガイドラインの詳細については、[第 7 章「HDL コーディング手法」](#)を参照してください。
- ・ 構文規則も含めた Verilog User Defined Primitive (UDP) の詳細は、Verilog 言語リファレンスのマニュアルを参照してください。

UDP の定義とインスタンス化

- ・ UDP は、インスタンス化前に定義しておく必要があります。
- ・ UDP の定義 :
 - primitive および endprimitive キーワードの間に記述します。
 - module-endmodule セクションの範囲外のどこかで検出される可能性があります。
- ・ UDP はゲートレベル プリミティブとユーザー定義モジュールと同じようにインスタンス化されます。

組み合わせ UDP

組み合わせ UDP は、出力の次の値を決定するために入力の値を使用して、組み合わせファンクションを記述できるようにします。

組み合わせ UDP のコード例

```
//
// Description and instantiation of a user defined primitive
// combinatorial function
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: Verilog_Language_Support/user_defined_primitives/udp_combinatorial_1.v
//
primitive myand2 (o, a, b);
    input  a, b;
    output o;

    table
        // a b : o
        0 0 : 0;
        0 1 : 0;
        1 0 : 0;
        1 1 : 1;
    endtable

endprimitive

module udp_combinatorial_1 (a, b, c, o);
    input  a, b, c;
    output o;

    wire  s;

    myand2 i1 (.a(a), .b(b), .o(s));
    myand2 i2 (.a(s), .b(c), .o(o));

endmodule
```

順次 UDP

順次 UDP は、入力値と出力の現在値を使用して、出力の次の値を決定します。

- ・ 順次 UDP を使用すると、次が実行できます。
 - レベル センシティブとエッジ センシティブ ビヘイビア両方を記述
 - フリップフロップおよびラッチなどの順次エレメントを記述
- ・ 初期値は、指定できます。

順次 UDP のコード例

```
//
// Description and instantiation of a user defined primitive
// Sequential function
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: Verilog_Language_Support/user_defined_primitives/udp_sequential_2.v
//
primitive mydff (q, d, c);
    input    c, d;
    output reg q;

    initial q = 1'b0;

    table
    // c d : q : q+
        r 0 : ? : 0;
        r 1 : ? : 1;
        f ? : ? : -;
        ? * : ? : -;
    endtable

endprimitive

module udp_sequential_2 (clk, si, so);
    input  clk, si;
    output so;

    wire  s1, s2;

    mydff i1 (.c(clk), .d(si), .q(s1));
    mydff i2 (.c(clk), .d(s1), .q(s2));
    mydff i3 (.c(clk), .d(s2), .q(so));

endmodule
```

Verilog の予約語

アスタリスク (*) の付いた単語は、Verilog の予約語ですが、XST でサポートされていません。

always	and	assign	automatic
begin	buf	bufif0	bufif1
case	casex	casez	cell*
cmos	config*	deassign	default
defparam	design*	disable	edge
else	end	endcase	endconfig*
endfunction	endgenerate	endmodule	endprimitive
endspecify	endtable	endtask	event
for	force	forever	fork
function	generate	genvar	highz0
highz1	if	ifnone	incdir*
include*	initial	inout	input
instance*	integer	join	large
liblist*	library*	localparam	macromodule
medium	module	nand	negedge
nmos	nor	noshow-cancelled*	not
notif0	notif1	or	output
parameter	pmos	posedge	primitive
pull0	pull1	pullup	pulldown
pulsestyle- _ondetect*	pulsestyle- _onevent*	rcmos	real
realtime	reg	release	repeat
rnmos	rpmos	rtran	rtranif0
rtranif1	scalared	show-cancelled*	signed
small	specify	specparam	strong0
strong1	supply0	supply1	table
task	time	tran	tranif0
tranif1	tri	tri0	tri1
triand	trior	trireg	use*
vectored	wait	wand	weak0
weak1	while	wire	wor
xnor	xor		

Verilog ビヘイビアー記述

XST では、特に記述のない限り Verilog ビヘイビアー記述がサポートされます。

Verilog ビヘイビアー記述の変数

- ・ Verilog ビヘイビアー記述の変数は、次のように宣言します。
 - integer
 - real
- ・ これらの宣言はテストコードでのみ使用されます。実際のハードウェア記述では、reg や wire などのデータ型を使用できます。
- ・ reg と wire の違いは、変数の値が reg では手続き代入文で、wire では継続代入文で指定される点です。
 - どちらもデフォルトの幅は 1 ビット (スカラ) です。
 - reg または wire 宣言で N ビット幅 (ベクタ) を指定するには、[] 内に左のビット位置と右のビット位置をコロンで区切って示します。
 - Verilog 2001 では、reg および wire データ型のどちらも符号付きまたは符号なしにできます。

変数宣言のコード例

```
reg [3:0] arb_priority;  
wire [31:0] arb_request;  
wire signed [8:0] arb_signed;
```

初期値

Verilog-2001 では、レジスタを宣言する際に初期化できます。

- ・ 指定される初期値は、次の通りです。
 - 定数値です。
 - 以前の初期値に依存できません。
 - 関数またはタスク呼び出しは使用できません。
 - レジスタに伝搬するパラメーター値にできます。
 - ベクタのすべてのビットを指定します。
- ・ 宣言部でレジスタの初期値を指定した場合、グローバル リセット時または電源投入時にレジスタの出力が指定した値に初期化されます。
- ・ この方法で指定された初期値は次のようになります。
 - レジスタの INIT 属性として NGC ファイルに渡されます。
 - ローカル リセットには依存しません。

初期値のレジスタへの代入

レジスタにセット/リセット時の初期値を指定できます。

- ・ レジスタのリセット ラインが最適な値になったときにレジスタに値を代入するには、次の例のように記述します。次の例を参照してください。
- ・ 変数に初期値を代入すると、値は次のようになります。
 - 出力がローカル リセットで制御されるフリップフロップとしてインプリメントされる
 - NGC ファイルに FDP または FDC フリップフロップとして渡される

初期値のコード例 1

```
reg arb_onebit = 1'b0;  
reg [3:0] arb_priority = 4'b1011;
```

初期値のコード例 2

```
always @(posedge clk)  
begin  
    if (rst)  
        arb_onebit <= 1'b0;  
end
```

reg および wire の配列

Verilog では、reg および wire の配列を次の例のように定義できます。

配列のコード例 1

次のコード例は 32 エLEMENTの配列で、ELEMENTの幅はそれぞれ 4 ビットです。

```
reg [3:0] mem_array [31:0];
```

配列のコード例 2

次のコード例は 64 個の 8 ビット幅エレメントの配列で、構造記述の Verilog コードでのみ代入できます。

```
wire [7:0] mem_array [63:0];
```

多次元配列

XST では、2 次元までの多次元配列型がサポートされます。

- ・ 多次元配列は、次のいずれかにできます。
 - すべてのネット
 - すべての変数データ型
- ・ 配列を使用すると、コード代入や数値演算を行うことができます。
- ・ 一度に選択できる配列のエレメントは 1 つのみです。
- ・ 次には、多次元配列を使用することはできません。
 - システム タスクまたは関数
 - 通常タスクまたは関数

多次元配列の Verilog コード例 1

次のコード例は 256 x 16 ワイヤー エレメント (各 8 ビット) の配列で、構造記述の Verilog コードでのみ代入できます。

```
wire [7:0] array2 [0:255][0:15];
```

多次元配列の Verilog コード例 2

次のコード例は 256 x 8 レジスタ エレメント (各 64 ビット幅) の配列で、ビヘイビア記述の Verilog コードでのみ代入できます。

```
reg [63:0] regarray2 [255:0][7:0];
```

データ型

Verilog のビット データ型には、次の 4 つの値があります。

- ・ **0**
論理値 0
- ・ **1**
論理値 1
- ・ **x**
不定値
- ・ **z**
ハイ インピーダンス

XST でサポートされる Verilog データ型

- ・ net
- ・ wire
- ・ tri
- ・ triand/wand
- ・ trior/wor
- ・ registers
- ・ reg
- ・ integer
- ・ supply nets
- ・ supply0
- ・ supply1
- ・ constants
- ・ parameter
- ・ 多次元配列 (メモリ)

ネットおよびレジスタ

ネットおよびレジスタは次のいずれかにできます。

- ・ 単数ビット (スカラ)
- ・ 複数ビット (ベクタ)

Verilog ビヘイビア記述のデータ型のコード例

Verilog モジュールの宣言セクションで使用する Verilog データ型の例を次に示します。

```
wire net1; // single bit net
reg r1; // single bit register
tri [7:0] bus1; // 8 bit tristate bus
reg [15:0] bus1; // 15 bit register
reg [7:0] mem[0:127]; // 8x128 memory register
parameter state1 = 3'b001; // 3 bit constant
parameter component = "TMS380C16"; // string
```

使用可能な文

XST では、Verilog ビヘイビア記述で使用可能な文がサポートされます。

- ・ 次に、使用可能な文 (変数および信号代入) を示します。
 - 変数 = 論理式
 - if (条件) 文
 - else 文
 - case (論理式)

```
expression: statement
...
default: statement
endcase
```
 - for (変数 = 論理式; 条件; 変数 = 変数 + 論理式) 文
 - while (条件) 文
 - forever 文
 - function および task
- ・ すべての変数は、integer (整数) または reg (レジスタ) として宣言されます。
- ・ 変数は wire として宣言することはできません。

論理式

Verilog ビヘイビア記述の論理式 :

- ・ 定数
- ・ 次の演算子の付いた変数
 - 数値演算
 - 論理演算
 - ◆ ビット単位の演算
 - ◆ 論理演算
 - 関係
 - 条件

論理演算子

論理演算は、次のいずれに適用されるかで、ビットごとまたは論理にさらに分類されます。

- ・ 複数のビットを使用する式
- ・ 1 つのビット

サポートされる演算子

演算	論理演算	関係演算	条件演算
+	&	<	?
-	&&	==	
*		===	
**		<=	
/	^	>=	
%	~	>=	
	~~	!=	
	~~	!==	
	<<	>	
	>>		
	<<<		
	>>>		

サポートされる論理式

論理式	シンボル	ステータス
連接	{ }	サポートあり
複製	{ }	サポートあり
演算	+, -, *, **	サポートあり
分周	/	次の場合のみサポートされます。 ・ 2 番目のオペランドが 2 のべき乗の場合 または ・ どちらのオペランドも定数の場合
剰余	%	2 番目のオペランドが 2 のべき乗の場合のみサポートあり
加算	+	サポートあり
減算	-	サポートあり
乗算	*	サポートあり

論理式	シンボル	ステータス
電力	**	サポートあり <ul style="list-style-type: none"> ・ 2 番目のオペランドが負でない場合は、両方のオペランドが定数であることが必要 ・ 最初のオペランドが 2 の場合は、2 番目のオペランドに変数を使用可能 ・ 実数データ型はサポートされず、結果が実数となるようなオペランドの組み合わせを使用するとエラーが発生する ・ X (不明) および Z (ハイインピーダンス) は使用不可
関係演算	>, <, >=, <=	サポートあり
論理否定	!	サポートあり
論理 AND	&&	サポートあり
論理 OR		サポートあり
論理等号	==	サポートあり
論理不等号	!=	サポートあり
ケース等号	===	サポートあり
ケース不等号	!==	サポートあり
ビットごとの否定	~	サポートあり
ビットごとの AND	&	サポートあり
ビットごとの内包的 OR		サポートあり
ビットごとの排他的 OR	^	サポートあり
ビットごとの等価	~, ^^	サポートあり
リダクション AND	&	サポートあり
リダクション NAND	~&	サポートあり
リダクション OR		サポートあり
リダクション NOR	~	サポートあり
リダクション XOR	^	サポートあり
リダクション XNOR	~, ^^	サポートあり
左シフト	<<	サポートあり
符号付き右シフト	>>>	サポートあり
符号付き左シフト	<<<	サポートあり
右シフト	>>	サポートあり
条件演算	?:	サポートあり
イベント OR	or, ', '	サポートあり

論理式の評価

次の表の === と != は、次のような特徴があります。

- ・ 特殊な比較演算子です。
- ・ シミュレーションで使用して、変数に値 x または z が代入されているかを確認します。
- ・ 合成では、これらの演算子は == および != として処理されます。

最も使用される演算子に基づいた評価済み論理式

a b	a==b	a===b	a!=b	a!=b	a&b	a&&b	a b	a b	a^b
0 0	1	1	0	0	0	0	0	0	0
0 1	0	0	1	1	0	0	1	1	1
0 x	x	0	x	1	0	0	x	x	x
0 z	x	0	x	1	0	0	x	x	x
1 0	0	0	1	1	0	0	1	1	1
1 1	1	1	0	0	1	1	1	1	0
1 x	x	0	x	1	x	x	1	1	x
1 z	x	0	x	1	x	x	1	1	x
x 0	x	0	x	1	0	0	x	x	x
x 1	x	0	x	1	x	x	1	1	x
x x	x	1	x	0	x	x	x	x	x
x z	x	0	x	1	x	x	x	x	x
z 0	x	0	x	1	0	0	x	x	x
z 1	x	0	x	1	x	x	1	1	x
z x	x	0	x	1	x	x	x	x	x
z z	x	1	x	0	x	x	x	x	x

ブロック

XST では、ブロック文が一部サポートされています。

- ・ ブロック文には、次のような特徴があります。
 - 複数の文をグループ化
 - begin と end キーワード間
 - ブロック内では、記述された順に文が実行される
- ・ XST では、順次ブロックのみがサポートされています。
- ・ XST では、パラレル ブロックはサポートされません。
- ・ ブロック内の手続き文は、すべてモジュール内で定義します。
- ・ 手続き型ブロックには次の 2 種類があります。
 - initial ブロック
 - always ブロック
- ・ 各ブロックは begin で開始し end で終了します。initial ブロックは合成では無視されるので、ここでは always ブロックのみを説明します。

- ・ always ブロックは通常、次のフォーマットで記述されます。各文は手続き代入文であり、セミコロンで区切られます。

```
always  
begin  
statement  
....  
end
```

モジュール

Verilog では、デザイン コンポーネントはモジュールで表されます。モジュールは宣言およびインスタンス化される必要があります。

モジュール宣言

- ・ Verilog ビヘイビア記述の module 宣言には、次が含まれます。
 - モジュール名
 - I/O ポートのリスト
 - 機能を定義するモジュール本体
- ・ module 文の終わりは、endmodule で示す必要があります。

I/O ポート

- ・ 回路の I/O ポートはモジュール宣言文で宣言します。
- ・ 各ポートでは次が指定されます。
 - 名前
 - モード :
 - ◆ 入力
 - ◆ 出力
 - ◆ 入出力
 - ポートが配列タイプの場合、範囲情報

Verilog ビヘイビア記述のモジュール宣言のコード例 1

```
module example (A, B, O);  
input  A, B;  
output O;  
  
assign O = A & B;  
  
endmodule
```

Verilog ビヘイビア記述のモジュール宣言のコード例 2

```
module example (  
    input  A,  
    input  B  
    output O  
):  
  
    assign O = A & B;  
  
endmodule
```

モジュール インスタンス化

- Verilog ビヘイビア記述のモジュール インスタンス化文には、次の特徴があります。
 - インスタンス名を定義します。
 - ポート関連リストが含まれます。
 - ポート関連リストでは、インスタンスが親モジュールでどのように接続されるかが指定されます。
 - リストの要素は、それぞれモジュール宣言のフォーマル ポートを親モジュールの実際のネットに関連付けられています。
- Verilog ビヘイビア記述のモジュールは、別のモジュールにインスタンス化されます。次の例を参照してください。

Verilog ビヘイビア記述のモジュール インスタンス化のコード例

```
module top (A, B, C, O);
    input  A, B, C;
    output O;
    wire  tmp;

    example inst_example (.A(A), .B(B), .O(tmp));

    assign O = tmp | C;

endmodule
```

継続代入文

XST では、明示的および暗示的両方の継続代入文がサポートされます。

- 継続代入文は、組み合わせロジックを簡潔に記述するために使用します。
- XST では、継続代入文で指定した遅延や電流は無視されます。
- 継続代入文は、wire および tri データ型のみに使用可能です。

明示的継続代入文

assign 文を使用する継続代入では、既に宣言されたネットに対して assign キーワードの後に代入式を定義します。

```
wire mysignal;
...
assign mysignal = select ? b : a;
```

暗示的継続代入文

assign 文を使用しない継続代入では、宣言文で代入式を定義します。

```
wire misignal = a | b;
```

手続き代入文

- ・ Verilog ビヘイビア記述の手続き代入文 :
 - reg として宣言された変数に値を代入します。
 - always ブロック、タスク、関数で最初に使用されます。
 - レジスタおよび FSM コンポーネントを記述します。
- ・ XST では、次がサポートされます。
 - 組み合わせ関数
 - 組み合わせタスクおよび順次タスク
 - 組み合わせブロックおよび順次 always ブロック

組み合わせ always ブロック

組み合わせロジックは、Verilog のタイミング制御文を使用すると効率的に記述できます。

- ・ 遅延タイミング制御文 [#]
- ・ イベント コントロール タイミング制御文 [@]

遅延タイミング制御文

遅延タイミング制御文 [#] には、次のような特徴があります。

- ・ シミュレーションにのみ関係します。
- ・ 合成では無視されます。

イベントコントロール タイミング制御文

次は、@ を使用したイベントコントロールのタイミング制御文を使用した組み合わせロジックの記述について説明しています。

- ・ 組み合わせ always 文には、always @ の後にかっこで囲まれたセンシティビティリストがあります。
- ・ センシティビティリストにある信号の 1 つでイベント (値の変化またはエッジ) が発生すると、always ブロックの処理が実行されます。
- ・ センシティビティリストには、次を含めることができます。
 - if や case などの条件で使用するすべての信号
 - 代入文の右側の信号すべて
- ・ 信号のリストの代わりに () なしで @ を使用すると、上記のような always ブロックの信号でイベントが発生した場合に、always ブロックの処理が実行されます。
- ・ 組み合わせプロセス文では、if 文または case 文のすべての分岐で信号が明示的に代入されていない場合、最後の値を保持するためにラッチが作成されます。
- ・ ラッチを生成するには、組み合わせプロセスで代入された信号がそのプロセス文のすべての条件に対して明示的に代入されるようにしてください。
- ・ プロセス文には、次の文を含めることができます。
 - 変数代入文および信号代入文
 - if - else 文
 - case 文
 - for および while ループ文
 - 関数およびタスクの呼び出し

if-else 文

XST では、if - else 文がサポートされます。

- ・ if-else 文では、真偽条件 (true-false) によって実行される文が決定されます。
 - 条件が真と判断された場合は if 文が実行されます。
 - 条件が偽 (または **x** か **z**) と判断された場合は else 文が実行されます。
- ・ キーワード begin と end を使用すると、複数文から成り立つブロックを実行できます。
- ・ If-else 文はネストさせることができます。

if-else 文のコード例

次のコード例は、if-else 文を使用してマルチプレクサを記述しています。

```
module mux4 (sel, a, b, c, d, outmux);
    input [1:0] sel;
    input [1:0] a, b, c, d;
    output [1:0] outmux;
    reg [1:0] outmux;

    always @(sel or a or b or c or d)
    begin
        if (sel[1])
            if (sel[0])
                outmux = d;
            else
                outmux = c;
        else
            if (sel[0])
                outmux = b;
            else
                outmux = a;
        end
    end
endmodule
```

case 文

XST では、case 文がサポートされています。

- ・ case 文は論理式を比較し、複数の並列分岐の 1 つを実行します。
 - 分岐は記述された順に評価され、
 - 最初に **true** になった分岐から実行されます。
 - 一致する分岐が見つからない場合は、デフォルトの分岐が実行されます。
- ・ case 文でサイズを指定していない整数を使用しないでください。必ず整数のサイズをビット数で指定しないと、結果が予測不可能になります。
- ・ casez は、分岐のすべてのビット位置の z 値をドントケアとして認識します。
- ・ casez は、分岐のすべてのビット位置の x と z の値をドントケアとして認識します。
- ・ casez または casex などの case 文では、疑問符 (?) もドントケアとして使用できます。

case 文を使用したマルチプレクサのコード例

```
module mux4 (sel, a, b, c, d, outmux);
    input [1:0] sel;
    input [1:0] a, b, c, d;
    output [1:0] outmux;
    reg [1:0] outmux;

    always @(sel or a or b or c or d)
    begin
        case (sel)
            2'b00: outmux = a;
            2'b01: outmux = b;
            2'b10: outmux = c;
            default: outmux = d;
        endcase
    end

endmodule
```

優先順処理の回避

- ・ 上記の例の case 文では、入力 sel の値が記述された優先順に評価されます。
- ・ この優先順処理を回避するには：
 - parallel_case という Verilog 属性を使用して、sel 入力が並列に評価されるようにします。
 - 上記の case 文は次のように置き換えることができます。

```
(* parallel_case *) case(sel)
```

for および repeat 文

XST では、for および repeat 文がサポートされています。

always ブロックでは、繰り返しまたはビット スライス構造を記述するのにも次のいずれかを使用できます。

- ・ **for** 文
- ・ **repeat** 文

for 文

for 文では、次のエレメントがサポートされます。

- ・ 定数の範囲
- ・ 次の演算子を使用したテスト条件の停止
 - <
 - <=
 - >
 - >=
- ・ 次のいずれかに適合する次ステップの計算
 - `var = var + step`
 - `var = var - step`
 - ◆ `var` はループ変数
 - ◆ `step` は定数値

repeat 文

- ・ `repeat` 文では定数値しか使用できません。
- ・ `disable` 文はサポートされていません。

```
module countzeros (a, Count);
    input [7:0] a;
    output [2:0] Count;
    reg [2:0] Count;
    reg [2:0] Count_Aux;

    integer i;

    always @(a)
    begin
        Count_Aux = 3'b0;
        for (i = 0; i < 8; i = i+1)
        begin
            if (!a[i])
                Count_Aux = Count_Aux+1;
        end
        Count = Count_Aux;
    end
endmodule
```


while ループ文

always ブロックでは、while ループ文を使用して繰り返し処理を実行できます。

- ・ while 文には、次の特徴があります。
 - テスト式が始めから false の場合は実行されません。
 - テスト式が偽 (false) になるまで、含まれる文を実行します。
- ・ 有効な Verilog の論理式であれば、どれでもテスト式として使用できます。
- ・ ループが永久に実行されるのを防ぐには、-loop_iteration_limit オプションを使用します。
- ・ while ループ文には disable 文を含めることができます。disable 文は、ラベルが付いているブロック内で使用します。

disable <blockname>

while ループ文のコード例

```
parameter P = 4;
always @(ID_complete)
begin : UNIDENTIFIED
    integer i;
    reg found;
    unidentified = 0;
    i = 0;
    found = 0;
    while (!found && (i < P))
    begin
        found = !ID_complete[i];
        unidentified[i] = !ID_complete[i];
        i = i + 1;
    end
end
end
```

順次 always ブロック

XST では、順次 always ブロックがサポートされています。

- ・ always ブロックと次のエッジトリガ イベント (posedge または negedge) を含むセンシティブティリストを使用して順次回路を記述します。
 - クロック イベント (必須)
 - オプションのセット/リセット イベント (非同期セット/リセット制御ロジックの記述)
- ・ オプションの非同期信号が記述されない場合、always ブロックは次のような構造になります。

```
always @(posedge CLK)
begin
    <synchronous_part>
end
```

- ・ オプションの非同期信号が記述される場合、always ブロックは次のような構造になります。

```
always @(posedge CLK or posedge ACTRL1 or a )
begin
    if (ACTRL1)
        <$asynchronous part>
    else
        <$synchronous_part>
end
```

順次 always ブロック コード例 1

次の例では、立ち上がりエッジクロックの付いた 8 ビットレジスタを記述しています。その他の制御信号はありません。

```
module seq1 (DI, CLK, DO);
    input [7:0] DI;
    input CLK;
    output [7:0] DO;
    reg [7:0] DO;

    always @(posedge CLK)
        DO <= DI ;
endmodule
```

順次 always ブロック コード例 2

次の例では、アクティブ High の非同期リセットを追加しています。

```
module EXAMPLE (DI, CLK, ARST, DO);
    input [7:0] DI;
    input CLK, ARST;
    output [7:0] DO;
    reg [7:0] DO;

    always @(posedge CLK or posedge ARST)
        if (ARST == 1'b1)
            DO <= 8'b00000000;
        else
            DO <= DI;

endmodule
```

順次 always ブロック コード例 3

このコード例では、次が記述されています。

- ・ アクティブ High の非同期リセット
- ・ アクティブ Low の非同期セット

```
module EXAMPLE (DI, CLK, ARST, ASET, DO);
    input [7:0] DI;
    input CLK, ARST, ASET;
    output [7:0] DO;
    reg [7:0] DO;

    always @(posedge CLK or posedge ARST or negedge ASET)
        if (ARST == 1'b1)
            DO <= 8'b00000000;
        else if (ASET == 1'b1)
            DO <= 8'b11111111;
        else
            DO <= DI;

endmodule
```

順次 always ブロックのコード例 4

このコード例では、次が記述されています。

- ・ 非同期セット/リセットを持たないレジスタ
- ・ 同期リセット

```
module EXAMPLE (DI, CLK, SRST, DO);
    input [7:0] DI;
    input CLK, SRST;
    output [7:0] DO;
    reg [7:0] DO;

    always @(posedge CLK)
        if (SRST == 1'b1)
            DO <= 8'b00000000;
        else
            DO <= DI;

endmodule
```

assign 文および deassign 文

XST では assign 文および deassign 文はサポートされません。

32 ビットを超える場合のビットの拡張

代入文の左側のビット幅が右側よりも大きい場合は、次のルールに従って、左側のビット幅が左にパディングされます。

- ・ 右側が符号付きの場合は、左側が符号付きビットでパディングされます。
- ・ 右側が符号なしの場合は、左側が 0 でパディングされます。
- ・ ビット指定のない x または z 定数の場合は、次の規則に従います。

右側の最上位ビットが z (ハイ インピーダンス) または x (不明) の場合、右側が符号付きまたは符号なしにかかわらず、左側にその値 (z または x) が追加されます。

タスクおよび関数

- ・ 同じコードを何度も使用する場合、タスクや関数を使用すると、次が可能になります。
 - コードの量を削減
 - 維持が容易
- ・ タスクおよび関数は、モジュール内で宣言して使用する必要があります。ヘッダー部には次のパラメーターが含まれます。
 - 入力パラメーター (関数の場合のみ)
 - 入力/出力/入出力パラメーター (タスクの場合)
- ・ 関数の戻り値は、符号付きまたは符号なしで宣言します。内容は組み合わせ always ブロック文に類似しています。

タスクおよび関数のコード例

アップデート情報は、「はじめに」のコード例を参照してください。

タスクおよび関数のコード例 1

```
//
// An example of a function in Verilog
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: Verilog_Language_Support/functions_tasks/functions_1.v
//
module functions_1 (A, B, CIN, S, COUT);
    input [3:0] A, B;
    input CIN;
    output [3:0] S;
    output COUT;
    wire [1:0] S0, S1, S2, S3;

    function signed [1:0] ADD;
        input A, B, CIN;
        reg S, COUT;
        begin
            S = A ^ B ^ CIN;
            COUT = (A&B) | (A&CIN) | (B&CIN);
            ADD = {COUT, S};
        end
    endfunction

    assign S0 = ADD (A[0], B[0], CIN),
           S1 = ADD (A[1], B[1], S0[1]),
           S2 = ADD (A[2], B[2], S1[1]),
           S3 = ADD (A[3], B[3], S2[1]),
           S = {S3[0], S2[0], S1[0], S0[0]},
           COUT = S3[1];

endmodule
```

タスクおよび関数のコード例 2

次は、同じ動作をタスクを使用して記述した例です。

```
//
// Verilog tasks
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: Verilog_Language_Support/functions_tasks/tasks_1.v
//
module tasks_1 (A, B, CIN, S, COUT);
    input [3:0] A, B;
    input CIN;
    output [3:0] S;
    output COUT;
    reg [3:0] S;
    reg COUT;
    reg [1:0] S0, S1, S2, S3;

    task ADD;
        input A, B, CIN;
        output [1:0] C;
        reg [1:0] C;
        reg S, COUT;
        begin
            S = A ^ B ^ CIN;
            COUT = (A&B) | (A&CIN) | (B&CIN);
            C = {COUT, S};
        end
    endtask

    always @(A or B or CIN)
    begin
        ADD (A[0], B[0], CIN, S0);
        ADD (A[1], B[1], S0[1], S1);
        ADD (A[2], B[2], S1[1], S2);
        ADD (A[3], B[3], S2[1], S3);
        S = {S3[0], S2[0], S1[0], S0[0]};
        COUT = S3[1];
    end

endmodule
```

再帰タスクおよび関数

Verilog-2001 では、再帰タスクおよび関数がサポートされます。

- ・ 再帰は、automatic キーワードだけで指定できます。
- ・ 再帰呼び出しが永久に実行されるのを防ぐために、繰り返す回数は 64 (デフォルト) に制限されています。
- ・ 回数を変更するには、-recursion_iteration_limit オプションを使用します。

再帰タスクおよび関数のコード例

```
function automatic [31:0] fac;
    input [15:0] n;
    if (n == 1)
        fac = 1;
    else
        fac = n * fac(n-1); //recursive function call
endfunction
```

定数関数

XST では、定数値を計算する関数呼び出しがサポートされます。

定数関数のコード例

```
//
// A function that computes and returns a constant value
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: Verilog_Language_Support/functions_tasks/functions_constant.v
//
module functions_constant (clk, we, a, di, do);
    parameter ADDRWIDTH = 8;
    parameter DATAWIDTH = 4;
    input clk;
    input we;
    input [ADDRWIDTH-1:0] a;
    input [DATAWIDTH-1:0] di;
    output [DATAWIDTH-1:0] do;

    function integer getSize;
        input addrwidth;
        begin
            getSize = 2**addrwidth;
        end
    endfunction

    reg [DATAWIDTH-1:0] ram [getSize(ADDRWIDTH)-1:0];

    always @(posedge clk) begin
        if (we)
            ram[a] <= di;
        end
        assign do = ram[a];
    end

endmodule
```

ブロッキングおよびノンブロッキング手続き代入文

ブロッキングおよびノンブロッキング手続き代入文には、タイミングを制御する要素が組み込まれています。

- ・ # および @ はタイミング制御文です。
- ・ 指定されたイベントが発生するまでその後に続く文は実行されません。
- ・ 合成では、# の遅延は無視されます。

ブロッキング手続き代入文のコード例 1

```
reg a;  
a = #10 (b | c);
```

ブロッキング手続き代入文のコード例 2

```
if (in1) out = 1'b0;  
else out = in2;
```

この代入の説明は次のとおりです。

- ・ プロセスに含まれる別の文が同時に実行されないようにします。
- ・ 主にシミュレーションで使用する必要があります。

ノンブロッキング手続き代入文のコード例 1

```
variable <= @(posedge_or_negedge_bit) expression;
```

- ・ ノンブロッキング代入文：
 - 文が実行されるときに式を評価します。
 - 同じプロセスに含まれるほかの文も同時に実行できます。
- ・ 変数は、指定された遅延後に変更されます。

ノンブロッキング手続き代入文のコード例 2

次に、ノンブロッキング手続き代入文の使用例を示します。

```
if (in1) out <= 1'b1;  
else out <= in2;
```

定数

定数は 10 進数の整数であると認識されますが、

- ・ 定数は、2 進、8 進、10 進、16 進に指定できます。
- ・ 定数を明示的に指定するには、適切な接頭辞を使用します。

定数式の例

次はすべて同じ値を表します。

- ・ 4'b1010
- ・ 4'o12
- ・ 4'd10
- ・ 4'ha

マクロ

- Verilog では、マクロが次のように定義されます。

```
'define TESTEQ1 4'b1101
```
- 定義されたマクロは、後で参照されます。

```
if (request == 'TESTEQ1)
```
- Verilog の **'ifdef** および **'endif** 文では、次を指定します。
 - マクロが定義されたかどうかを決定
 - 条件付きコンパイルを定義
- 'ifdef** で呼び出されたマクロが定義されている場合、そのコードはコンパイルされます。
 - マクロが定義されていない場合は、**'else** コマンドに続くコードがコンパイルされます。
 - 'else** は必須ではありませんが、条件文の最後に **'endif** を付ける必要があります。
- Verilog マクロ** コマンド ライン オプションを使用し、Verilog マクロを定義または再定義します。
 - Verilog マクロを使用すると、ソース コードを変更しなくてもデザインを簡単に修正できます。
 - Verilog マクロは、IP コアの生成やフロー テストで使用すると便利です。

マクロのコード例 1

```
'define myzero 0
assign mysig = 'myzero;
```

マクロのコード例 2

```
'ifdef MYVAR
module if_MYVAR_is_declared;
...
endmodule
'else
module if_MYVAR_is_not_declared;
...
endmodule
'endif
```

include ファイル

Verilog では、HDL ソース コードを複数のファイルに分割できます。

- 次のいずれかの方法で、さらに別のファイルを参照します。
 - ファイル インクルード方法
 - デザイン プロジェクト ファイル方法
- ザイリンクスでは、デザイン プロジェクト ファイルを使用する方法を推奨しています。

ファイル インクルード方法

ザイリンクスでは、このファイル インクルード方法は推奨していません。

- ・ 別のファイルに含まれるコードを参照するには、次の構文を使用します。
`'include "path/file-to-be-included"`
- ・ 相対パスまたは絶対パスのどちらでも使用できます。
- ・ 同じ Verilog ファイルに複数の 'include 文を含めることができます。こうすることで、複数ファイルで複数モジュールを記述するようなチーム デザイン環境で、コードが管理しやすくなります。
- ・ 'include 文で指定したファイルを認識させるには、ISE® Design Suite または XST にこのファイルのディレクトリを認識させる必要があります。
 - ファイルをプロジェクト ディレクトリに追加します。
ISE Design Suite では、デフォルトでプロジェクト ディレクトリが検索されます。
 - 'include 文に相対パスまたは絶対パスを含めます。
このパスにより、ISE Design Suite でプロジェクト ディレクトリ以外のディレクトリが指定できます。
 - 「Verilog の 'include ディレクトリの指定 (-vlgincdir)」を参照してください。
このオプションにより、XST で直接 include ファイル ディレクトリが指定されます。
- ・ デザイン階層を構築するのに 'include ファイルが必要とされる場合、ファイルは次のいずれかの方法で指定しておく必要があります。
 - プロジェクト ディレクトリに含める
 - 相対パスまたは絶対パスで参照する。ファイルをプロジェクトに追加する必要はありません。

デザイン プロジェクト ファイル方法

ザイリンクスでは、デザイン プロジェクト ファイルを使用する方法を推奨しています。

- ・ Verilog ファイルを残りのプロジェクトで認識させるには、XST デザイン プロジェクト ファイルでそのファイルをリストします。
- ・ ファイル インクルード方法を使用して Verilog ファイルを含めた場合は、XST デザイン プロジェクト ファイルにそのファイルをリストしないでください。リストすると、次のようなエラー メッセージが表示されます。

```
ERROR:HDLCompiler:687 - "include_sub.v" Line 1: Illegal  
redeclaration of module <sub>.
```
- ・ このエラー メッセージは、Verilog ファイルを複数のインクルード方法でプロジェクトに追加した場合に表示されることがあります。これは、ISE Design Suite がこれらのファイルを XST デザイン プロジェクト ファイルに自動的に追加するために、定義が重複してしまうからです。

Verilog ビヘイビア記述のコメント

Verilog ビヘイビア記述のコメント方法は、C++ のようなプログラミング言語と同様です。

1 行のコメント

コメントが 1 行の場合は // で開始します。

```
// This is a one-line comment.
```

複数行のブロックのコメント

コメントが複数行になる場合は /* で開始して */ で終わるようにその部分を囲みます。

```
/* This is a  
multiple-line  
comment.  
*/
```

generate 文

Verilog ビヘイビア記述の generate 文 :

- ・ 次が作成できます。
 - パラメーター変更可能なスケーラブルなコード
 - 反復的な構文やスケーラブルな構文
 - 特定の条件を満たす条件関数
- ・ Verilog のエラボレーション中に実行されます。
- ・ 条件別にデザインにインスタンスエートできます。
- ・ モジュール範囲内で記述します。
- ・ generate キーワードで開始します。
- ・ endgenerate キーワードで終了します。

generate 文を使用して作成した構造

generate 文では、次のような構造が作成できます。

- ・ プリミティブまたはモジュールのインスタンス
- ・ initial または always 手続きブロック
- ・ 継続代入文
- ・ ネットおよび変数の宣言
- ・ パラメーターの再定義
- ・ タスクまたは関数の定義

サポートされる generate 文

XST では、Verilog ビヘイビア記述の generate 文がサポートされます。

- ・ generate ループ文 (generate-for)
- ・ generate 条件文 (generate-if-else)
- ・ generate ケース文 (generate-case)

generate ループ文

generate-for ループ文を使用するとは、モジュール内に 1 つ以上のインスタンスが作成されます。

generate-for ループ文は for ループ文と同様に使用できますが、次のような制限があります。

- ・ generate-for ループ文のインデックスには、genvar 変数を使用する必要があります。
- ・ for ループ制御内の代入は、genvar 変数を参照する必要があります。
- ・ for ループ文の内容は begin 文と end 文で囲み、
- ・ begin 文には固有の修飾子が付いた名前を使用します。

generate ループ文を使用した 8 ビット加算器のコード例

```
generate
genvar i;
    for (i=0; i<=7; i=i+1)
    begin : for_name
        adder add (a[8*i+7 : 8*i], b[8*i+7 : 8*i], ci[i], sum_for[8*i+7 : 8*i], c0_or[i+1]);
    end
endgenerate
```

generate 条件文

generate-if-else 文は、オブジェクトの生成を条件で制御するために使用します。

- ・ if-else 文の各分岐は begin 文と end 文で囲みます。
- ・ begin 文には固有の修飾子が付いた名前を使用します。

generate 条件文のコード例

次の例では、データワードの幅に基づいて 2 つの異なるインプリメンテーションで乗算器をインスタンス化しています。

```
generate
    if (IF_WIDTH < 10)
    begin : if_name
        multiplier_imp1 # (IF_WIDTH) u1 (a, b, sum_if);
    end
    else
    begin : else_name
        multiplier_imp2 # (IF_WIDTH) u2 (a, b, sum_if);
    end
endgenerate
```

generate case 文

generate-case 文は、オブジェクトの生成をさまざまな条件で制御するために使用します。

- ・ generate-case 文の各分岐の内容は begin 文と end 文で囲みます。
- ・ begin 文には固有の修飾子が付いた名前を使用します。

generate-case 文の Verilog ビヘイビアー記述のコード例

次の例では、データワードの幅に基づいて 2 つ以上のインプリメンテーションで乗算器をインスタンス化しています。

```
generate
  case (WIDTH)
    1:
      begin : case1_name
        adder #(WIDTH*8) x1 (a, b, ci, sum_case, c0_case);
      end
    2:
      begin : case2_name
        adder #(WIDTH*4) x2 (a, b, ci, sum_case, c0_case);
      end
    default:
      begin : d_case_name
        adder x3 (a, b, ci, sum_case, c0_case);
      end
  endcase
endgenerate
```


混合言語のサポート

XST では、特に記述のない限り、VHDL と Verilog の混合言語プロジェクトがサポートされます。

VHDL と Verilog の混合

- ・ VHDL と Verilog の混合は、デザイン ユニット (セル) のインスタンス化に制限されています。
 - Verilog モジュールは VHDL コードからインスタンス化できます。
 - VHDL エンティティは Verilog コードからインスタンス化できます。
 - それ以外の方法で VHDL と Verilog は混合できません。たとえば、Verilog ソースコードを直接 VHDL ソースコードに埋め込むことはできません。
- ・ VHDL デザインでは、VHDL タイプ、ジェネリック、およびポートの制限されたサブセットを Verilog モジュールとの境界に使用できます。
- ・ Verilog デザインでは、Verilog タイプ、ジェネリック、およびポートの制限されたサブセットを VHDL モジュールまたはコンフィギュレーションとの境界に使用できます。
- ・ XST では、エラレーション段階で VHDL デザイン ユニットが Verilog モジュールにバインドされます。
- ・ プロジェクトを構成する VHDL および Verilog ファイルは、独自の HDL プロジェクトファイルで指定します。詳細は、第 2 章「XST プロジェクトの作成および合成」を参照してください。

インスタンス化

- ・ Verilog モジュールを VHDL デザイン ユニットにバインドする際は、デフォルトのバインド方法に基づくコンポーネント インスタンス化が使用されます。
- ・ VHDL に Verilog モジュールをインスタンス化する場合、XST では次はサポートされていません。
 - コンフィギュレーション指定
 - ダイレクト インスタンス化
 - コンポーネント コンフィギュレーション

VHDL および Verilog のライブラリ

- ・ VHDL および Verilog ライブラリが論理的に統一されます。
- ・ コンパイル用のデフォルトの作業ディレクトリ (xsthdpdir) は、VHDL でも Verilog でも使用できます。
- ・ xhdp.ini のメカニズムには、次のような特徴があります。
 - － 論理ライブラリ名をホスト ファイル システムの物理ディレクトリ名にマップ
 - － VHDL でも Verilog でも使用可能
- ・ デザイン ユニット (セル) を統一された論理ライブラリで検索するための検索順を指定できます。エラボレーションの段階でこの検索順に従って、VHDL エンティティまたは Verilog モジュールが検索され、混合言語プロジェクトにバインドされます。

VHDL/Verilog の境界規則

VHDL と Verilog の境界は、デザイン ユニットのレベルにより決定します。

- ・ VHDL のエンティティまたはアーキテクチャには Verilog モジュールをインスタンスエートできます。詳細は、「[Verilog への VHDL のインスタンスエート](#)」を参照してください。
- ・ Verilog のモジュールには VHDL エンティティをインスタンスエートできます。詳細は、「[VHDL への Verilog のインスタンスエート](#)」を参照してください。

Verilog への VHDL のインスタンスエート

Verilog デザインへ VHDL デザイン ユニットをインスタンスエートするには、次の手順に従ってください。

1. インスタンスエートする VHDL エンティティと同じモジュール名（アーキテクチャ名を付けたものも可）を宣言
2. 通常の Verilog インスタンスエーションを実行

XST の制限 (Verilog 内に VHDL の場合)

Verilog モジュールから VHDL デザイン ユニットをインスタンスエートする場合、XST では次のような制限があります。

- ・ Verilog デザインにインスタンスエートできる VHDL の構文は、VHDL エンティティのみです。
 - その他の VHDL の構文は Verilog コードで認識されません。
 - XST では、エンティティ/アーキテクチャ ペアが Verilog と VHDL の境界として使用されます。
- ・ ポートの関連付けは明示的に行う必要があります。ポート マップでは、必ず正式な有効ポート名を指定してください。
- ・ パラメーターは、値が変化しない場合でも、インスタンスエート時にすべて渡す必要があります。
- ・ パラメーターを変更する場合は、どのパラメーターかを指定する必要があります。順序は認識されません。この場合、defparam を使用するのではなくインスタンスエーションを使用してください。

XST のバインド

XST では、バインド処理はエラボレーション段階で行われます。バインド中は、次が実行されます。

1. XST は次からインスタンスエートされたモジュールと同じ名前の Verilog モジュールを検索します。
 - a. 未定義の論理ライブラリのユーザー指定リスト
 - b. ユーザー指定順詳細は「[LSO の規則](#)」を参照してください。
2. モジュール インスタンスエーション文で指定したアーキテクチャ名は無視されます。
3. Verilog モジュールが見つかった場合は、その名前がバインドされます。
4. XST で Verilog モジュールが検出されない場合は、次が実行されます。
 - ・ Verilog モジュールが VHDL エンティティとして処理されます。
 - ・ XST は次から大文字と小文字を区別して名前的一致する最初の VHDL エンティティを検索します。
 - a. 未定義の論理ライブラリのユーザー指定リスト
 - b. ユーザー指定順

注記：これにより、VHDL デザイン ユニットは拡張指示子付きで格納されたと認識されます。

XST の制限 (VHDL から Verilog の場合)

Verilog モジュールから VHDL デザイン ユニットをインスタンスエートする場合、XST では次のような制限があります。

- ・ ポートの関連付けは明示的に行う必要があります。ポート マップでは、必ず正式な有効ポート名を指定してください。
- ・ パラメーターは、値が変化しない場合でも、インスタンスエート時にすべて渡す必要があります。
- ・ パラメーターを変更する場合は、どのパラメーターかを指定する必要があります。順序は認識されません。この場合、defparam を使用するのではなくインスタンスエーションを使用してください。

使用可能なコード例

```
ff #(.init(2'b01)) ul (.sel(sel), .din(din), .dout(dout));
```

使用不可なコード例

```
ff ul (.sel(sel), .din(din), .dout(dout));  
defparam ul.init = 2'b01;
```

VHDL への Verilog のインスタンスシート

- ・ VHDL デザインに Verilog モジュールをインスタンスシートするには、次の手順に従います。
 1. インスタンスシートする Verilog モジュールと同じ名前の VHDL コンポーネントを宣言します。
 2. 大文字/小文字の区別を監視します。
 3. Verilog モジュール名がすべて小文字でない場合は、case 文を使用してモジュールの大文字/小文字を保持するように設定します。
 - ISE® Design Suite
[Synthesize - XST] プロセスから [Process Properties] ダイアログ ボックスを表示して、[Synthesis Options] → [Case] → [Maintain] を選択
 - コマンド ライン
-case を maintain に設定
 4. VHDL コンポーネントをインスタンスシートするのと同様に、Verilog コンポーネントをインスタンスシートします。
- ・ VHDL コンフィギュレーション宣言を使用して、このコンポーネントを特定のライブラリからの特定のデザイン ユニットのバインドする方法はサポートされていません。サポートされるのは、デフォルト Verilog モジュールのバインドのみです。
- ・ VHDL デザインにインスタンスシートできる Verilog の構文は、Verilog モジュールのみです。その他の Verilog の構文は VHDL コードで認識されません。
- ・ エラレーションの段階で、デフォルトのバインド処理が行われるすべてのコンポーネントは、対応するコンポーネントの名前と同じ名前のデザイン ユニットとして処理されます。
- ・ バインド段階では、コンポーネント名は VHDL デザイン ユニット名として扱われ、work という論理ライブラリ内で検索されます。
 - VHDL デザイン ユニットが見つかった場合、バインドされます。
 - XST で VHDL デザイン ユニットが検出されない場合は、次が実行されます。
 - ◆ コンポーネント名が Verilog モジュール名として処理されます。
 - ◆ 大文字/小文字を区別した検索が実行されます。
- ・ Verilog モジュールは、統一された論理ライブラリから指定したライブラリの検索順に検索されます。詳細は、「[Library Search Order \(LSO\) ファイル](#)」を参照してください。
- ・ XST では、最初に一致した Verilog モジュールを選択してバインドします。
- ・ ライブラリは統一されているため、VHDL デザイン ユニットと同じ名前の Verilog セルは同じ論理ライブラリに共存させることはできません。
- ・ 同じ名前のセル/ユニットが新しくコンパイルされると、以前にコンパイルされたものが上書きされます。

ジェネリックのサポート

XST では、混合言語デザインで次の VHDL ジェネリック タイプがサポートされます。

- ・ integer
- ・ real
- ・ string
- ・ boolean

ポート マップ

XST では、次がサポートされます。

- ・ Verilog にインスタンス化された VHDL のポート マップ
- ・ VHDL にインスタンス化された Verilog のポート マップ

Verilog にインスタンス化された VHDL のポート マップ

VHDL エンティティが Verilog モジュールにインスタンス化される場合、ポートの詳細は次のようになります。

- ・ サポートされる方向：
 - in
 - out
 - inout
- ・ サポートされない方向：
 - buffer
 - linkage
- ・ 使用可能なデータ型：
 - bit
 - bit_vector
 - std_logic
 - std_ulogic
 - std_logic_vector
 - std_ulogic_vector

VHDL にインスタンス化された Verilog のポート マップ

Verilog モジュールが VHDL エンティティまたはアーキテクチャにインスタンス化される場合、ポートの詳細は次のようになります。

- ・ サポートされる方向：
 - input
 - output
 - inout
- ・ 使用可能なデータ型：
 - wire
 - reg
- ・ XST では、次がサポートされません。
 - Verilog の双方向パス オプションへの接続
 - 混合言語の境界に名前のない Verilog ポート

大文字と小文字が混合している Verilog モジュールのポート名を接続する場合は、コンポーネント宣言と同じようにしてください。Verilog ポート名はすべて小文字であると判断されます。

LSO ファイル

ライブラリ検索順ファイル (LSO) では、VHDL/Verilog 混合デザインに対して XST で使用するライブラリの検索順が指定されます。

- ・ ファイルはプロジェクト ファイルに現れる順序で検索されます。
- ・ XST では、次の場合デフォルトの検索順が使用されます。
 - LSO ファイルに DEFAULT_SEARCH_ORDER キーワードが含まれる場合
 - LSO ファイルが指定されていない場合

ISE Design Suite での LSO ファイルの指定

Library Search Order (LSO) ファイルのデフォルト名は `project_name.lso` です。

- ・ `project_name.lso` が存在する場合はそれが保持され、そのまま使用されます。
- ・ `project_name.lso` ファイルが存在しない場合は、次が実行されます。
 - デフォルトの `project_name.lso` ファイルを作成
 - ファイルの最初の行に DEFAULT_SEARCH_ORDER キーワードを記述
- ・ プロジェクト名が最上位レベルのブロックの名前になります。

コマンドライン モードでの LSO ファイルの指定

- ・ LSO (ライブラリ検索順) ファイルを指定するには、LSO コマンドライン オプション (`-lso`) を使用します。
- ・ `-lso` オプションを使用しない場合は、LSO ファイルなしでデフォルトのライブラリ検索順が使用されます。

LSO の規則

XST では、混合言語プロジェクトを処理する際、Library Search Order (LSO) ファイルの内容別に次の検索順規則が使用されます。

- ・ [空の LSO ファイル](#)
- ・ [DEFAULT_SEARCH_ORDER キーワードのみの場合](#)
- ・ [DEFAULT_SEARCH_ORDER キーワードとライブラリ リストがある場合](#)
- ・ [ライブラリ リストのみの場合](#)
- ・ [DEFAULT_SEARCH_ORDER キーワードがなく、存在しないライブラリ名が使用される場合](#)

空の LSO ファイル

LSO ファイルが空の場合、XST では次が実行されます。

- ・ LSO ファイルが空であることを示す警告メッセージが表示されます。
- ・ デフォルトのライブラリ検索順を使用してプロジェクト ファイルで指定したファイルが検索されます。
- ・ プロジェクト ファイルに現れる順序で、ライブラリが LSO ファイルにリストされます。

DEFAULT_SEARCH_ORDER キーワードのみの場合

LSO ファイルに DEFAULT_SEARCH_ORDER キーワードのみが含まれる場合、次が実行されます。

- ・ プロジェクト ファイルに現れる順序でライブラリ ファイルが検索されます。
- ・ LSO ファイルから DEFAULT_SEARCH_ORDER キーワードが削除されます。
- ・ プロジェクト ファイルに現れる順序で、ライブラリが LSO ファイルにリストされます。

検索順の例

1. プロジェクト ファイル my_proj.prj には、次のような内容が含まれています。

```
vhdl      vllib1    f1.vhd
verilog   rtflllib   f1.v
vhdl      vllib2    f3.vhd
```

2. ISE® Design Suite で作成される LSO ファイル my_proj.lso の内容は、次のとおりです。

```
DEFAULT_SEARCH_ORDER
```

3. XST では、次の検索順が使用されます。

```
vllib1
rtflllib
vllib2
```

プロセス後、同じ内容がアップデートされた my_proj.lso に表示されます。

DEFAULT_SEARCH_ORDER キーワードとライブラリ リストがある場合

- ・ LSO ファイルに次が含まれる場合：
 - DEFAULT_SEARCH_ORDER キーワード、および
 - ライブラリ リスト
- ・ XST では、次が実行されます。
 - プロジェクト ファイルに現れる順序でライブラリ ファイルが検索されます。
 - LSO ファイルに含まれるライブラリのリストは無視されます。
 - LSO ファイルはアップデートされません。

検索順の例

1. プロジェクトファイル my_proj.prj には、次のような内容が含まれています。

```
vhdl      vllib1    f1.vhd
verilog   rtflllib   f1.v
vhdl      vllib2    f3.vhd
```

2. LSO ファイル my_proj.lso の内容は、次のとおりです。

```
rtflllib
vllib2
vllib1
DEFAULT_SEARCH_ORDER
```

3. XST では、次の検索順が使用されます。

```
vllib1
rtflllib
vllib2
```

4. プロセス後の my_proj.lso の内容に変更はありません。

```
rtflllib
vllib2
vllib1
DEFAULT_SEARCH_ORDER
```

ライブラリ リストのみの場合

LSO ファイルにライブラリのリストが含まれており、DEFAULT_SEARCH_ORDER キーワードがない場合、XST では次が実行されます。

- ・ LSO ファイルにリストされている順序でライブラリ ファイルが検索されます。
- ・ LSO ファイルはアップデートされません。

ファイル検索の例

1. プロジェクトファイル my_proj.prj には、次のような内容が含まれています。

```
vhdl vhlib1 f1.vhd  
verilog rtfllib f1.v  
vhdl vhlib2 f3.vhd
```

2. LSO ファイル my_proj.lso の内容は、次のとおりです。

```
rtfllib  
vhlib2  
vhlib1
```

3. XST では、次の検索順が使用されます。

```
rtfllib  
vhlib2  
vhlib1
```

4. プロセス後の my_proj.lso の内容は、次のとおりです。

```
rtfllib  
vhlib2  
vhlib1
```

DEFAULT_SEARCH_ORDER キーワードがなく、存在しないライブラリ名が使用される場合

LSO ファイルが次のような場合、XST でライブラリが無視されます。

- ・ プロジェクトまたは INI ファイルに存在しないライブラリ名が含まれていて、さらに
- ・ DEFAULT_SEARCH_ORDER キーワードが含まれない場合

検索順の例

1. プロジェクトファイル my_proj.prj には、次のような内容が含まれています。

```
vhdl vhlib1 f1.vhd  
verilog rtfllib f1.v  
vhdl vhlib2 f3.vhd
```

2. LSO ファイル my_proj.lso の内容は、次のとおりです。

```
personal_lib  
rtfllib  
vhlib2  
vhlib1
```

3. XST では、次の検索順が使用されます。

```
rtfllib  
vhlib2  
vhlib1
```

4. プロセス後の my_proj.lso の内容は、次のとおりです。

```
rtfllib  
vhlib2  
vhlib1
```

HDL コーディング手法

HDL コーディング手法を使用すると、次が実行できます。

- ・ デジタル ロジック回路でよく使用される機能を記述できます。
- ・ ザイリンクス デバイスのアーキテクチャ機能を利用できます。

ISE® Design Suite から合成テンプレートを使用する方法については、ISE Design Suite ヘルプを参照してください。

VHDL の利点

- ・ 規則はより厳格になり、明確に入力する必要があり、自由度は低く、エラーが発生しやすくなります。
- ・ HDL ソース コードでの RAM コンポーネントの初期化が簡単 (Verilog の初期ブロックの方が困難)
- ・ パッケージ サポート
- ・ カスタム タイプ
- ・ 列挙型
- ・ reg と wire を混乱することがない

Verilog の利点

- ・ System Verilog への拡張 (現在のところ XST でサポートされていません)
- ・ C 言語のような構文
- ・ コードは VHDL よりもコンパクト
- ・ コメントのブロック化 (複数行)
- ・ VHDL のようにコンポーネント インスタンス化が多くない

マクロ推論フローの概要

マクロ推論は、XST 合成フローでは次の 3 段階で発生します。

- ・ 基本的なマクロは HDL 合成中に推論されます。
- ・ 複雑なマクロはアドバンス HDL 合成中に推論されます。
- ・ ほかのマクロは、タイミング情報が使用可能になって、より多くの情報を含む決定が可能になったときに、下位レベルの最適化中に推論されます。
- ・ アドバンス HDL 合成中に推論されるマクロは、通常その前の HDL 合成中に推論された基本的なマクロが複数集まったものになります。ほとんどの場合、XST の推論エンジンでは階層の違いに関係なく、これらのグループ化が実行できます。ただし、[階層の維持 \(KEEP_HIERARCHY\)](#) が yes に設定されている場合は、例外です。

例

ブロック RAM は、あるユーザーが定義した階層ブロックに記述された RAM コアの機能と、別のユーザーが定義した階層で記述されたレジスタを一緒にまとめて推論されます。これにより、HDL プロジェクトをモジュラ方式で構成できるので、別の VHDL エンティティおよび Verilog モジュールで記述したデザイン エLEMENT 間の関係が XST で認識可能です。

- ・ 基本のビットレベルのエLEMENTそれぞれを別の階層に記述しないでください。
 - このように記述すると、合成ツールの RTL 推論機能が使用されなくなってしまいます。
 - HDL ソースコードの構造については、「[DSP の推論](#)」のデザイン プロジェクトを参照してください。

フリップフロップおよびレジスタ

- ・ XST では、次の制御信号付きのフリップフロップおよびレジスタが認識されます。
 - 立ち上がりエッジまたは立ち下がりエッジのクロック
 - 非同期セット/リセット
 - 同期セット/リセット
 - クロック イネーブル
- ・ フリップフロップおよびレジスタは、次を使用して記述されます。
 - 順次 process 文 (VHDL)
 - always ブロック (Verilog)
- ・ process または always ブロックのセンシティビティリストには、次がリストされている必要があります。
 - クロック信号
 - すべての非同期制御信号
- ・ 順次ロジックの記述方法については、次を参照してください。
 - [第 3 章「VHDL のサポート」](#)
 - [第 4 章「Verilog のサポート」](#)

フリップフロップおよびレジスタの初期化

回路に電源が投入されたときにレジスタの内容を初期化するには、それを表す信号のデフォルト値を指定します。

VHDL でのフリップフロップおよびレジスタの初期化

VHDL で回路に電源が投入されたときにレジスタの内容を初期化するには、次のように信号を宣言します。

```
signal example1 : std_logic := '1';
signal example2 : std_logic_vector(3 downto 0) := (others => '0');
signal example3 : std_logic_vector(3 downto 0) := "1101";
```

Verilog でのフリップフロップおよびレジスタの初期化

Verilog の場合、初期内容は次のように記述されます。

```
reg example1 = 'b1 ;
reg [15:0] example2 = 16'b1111111011011100;
reg [15:0] example3 = 16'hFEDC;
```

合成済みフリップフロップは、回路に電源が投入されたときにグローバル リセットがオンになると、ターゲット デバイスで指定した値に初期化されます。

フリップフロップおよびレジスタの制御信号

フリップフロップおよびレジスタの制御信号には、次が含まれます。

- ・ クロック
- ・ 非同期および同期のセット/リセット信号
- ・ クロック イネーブル

コード作成のガイドライン

- ・ 次は、コード作成のガイドラインです。
 - スライス ロジック使用率の最小化
 - 最高の回路パフォーマンス
 - ブロック RAM コンポーネントおよび DSP ブロックなどのデバイス リソースの使用
- ・ レジスタを非同期にセット/リセットしない
 - 制御セットのマッピングがやり直せなくなる
 - ブロック RAM コンポーネントおよび DSP ブロックなどのデバイス リソースの順次機能は同期にしかセットまたはリセットできません。
 - これらのリソースは使用できなくなるか、最適にコンフィギュレーションされなくなります。
 - 同期初期化を使用します。
- ・ コーディング ガイドラインでレジスタを非同期にセットまたはリセットにする必要がある場合は、[非同期から同期への変換 \(ASYNC_TO_SYNC\)](#) を使用します。これにより、同期 セット/リセットが使用できるようになります。
- ・ セットとリセット両方が付いたフリップフロップは記述できません。
 - セットとリセットの両方を含むフリップフロップ プリミティブは同期/非同期に関わらず、使用できなくなっています。
 - ツールで拒否されない場合、この組み合わせによりインプリメンテーションでエリアやパフォーマンスに悪影響がでることもあります。
- ・ 非同期リセットと非同期セットの両方を含むフリップフロップは記述しないでください。XST では、こういったフリップフロップが拒絶され、コスト的に同等のモデルへターゲット変更されることはありません。
- ・ できる限り、セット/リセット ロジックを使用しないでください。たとえば、回路のグローバル リセットを使って初期内容を定義するなど、その他のコストがより低くてすむような方法で、希望どおりの結果にできることがあります。
- ・ フリップフロップ プリミティブのクロック イネーブル、セット/リセット制御入力は常にアクティブ High で記述します。アクティブ Low にすると、その結果のインバータ ロジックにより回路のパフォーマンスが悪化します。

フリップフロップおよびレジスタの関連制約

- ・ I/O レジスタの IOB 内へのパック
- ・ レジスタの複製
- ・ 等価レジスタの削除
- ・ レジスタ自動調整
- ・ アーキテクチャ サポート

フリップフロップおよびレジスタのインプリメンテーションの制御方法詳細は、「[LUT へのロジックのマッピング](#)」を参照してください。

フリップフロップおよびレジスタのレポート

- ・ レジスタが推論されると、HDL 合成中にレポートが出力されます。
- ・ レジスタは、アドバンス HDL 合成が終了すると、個別のフリップフロップに展開されてレポートされます。
- ・ HDL 合成中に推論されたレジスタの数は、Design Summary セクションのフリップフロッププリミティブの数と完全には一致しないことがあります。
- ・ フリップフロップ プリミティブの数は、次のプロセスによって変わります。
 - DSP ブロックやブロック RAM コンポーネントへのレジスタの吸収
 - レジスタの複製
 - 定数や同等のフリップフロップの削除
 - レジスタ自動調整

フリップフロップおよびレジスタのレポート例

```
=====
*                               HDL Synthesis                               *
=====

Synthesizing Unit registers_5>.
  Found 4-bit register for signal Q>.
  Summary:
    inferred    4 D-type flip-flop(s).
Unit registers_5> synthesized.

=====

HDL Synthesis Report

Macro Statistics
# Registers                      : 1
  4-bit register                 : 1

=====

=====
*                               Advanced HDL Synthesis                       *
=====

(...)

=====

Advanced HDL Synthesis Report

Macro Statistics
# Registers                      : 4
  Flip-Flops                    : 4

=====
```

フリップフロップおよびレジスタのコード例

アップデート情報は、「はじめに」のコード例を参照してください。

フリップフロップおよびレジスタの VHDL コード例

```
--
-- Flip-Flop with
--     Rising-edge Clock
--     Active-high Synchronous Reset
--     Active-high Clock Enable
--     Initial Value
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/registers/registers_6.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity registers_6 is
    port(
        clk    : in  std_logic;
        rst    : in  std_logic;
        clken   : in  std_logic;
        D      : in  std_logic;
        Q      : out std_logic);
end registers_6;

architecture behavioral of registers_6 is
    signal S : std_logic := '0';
begin

    process (clk)
    begin
        if rising_edge(clk) then
            if rst = '1' then
                S <= '0';
            elsif clken = '1' then
                S <= D;
            end if;
        end if;
    end process;

    Q <= S;

end behavioral;
```

フリップフロップおよびレジスタの Verilog コード例

```
//
// 4-bit Register with
//     Rising-edge Clock
//     Active-high Synchronous Reset
//     Active-high Clock Enable
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/registers/registers_6.v
//
module v_registers_6 (clk, rst, clken, D, Q);
    input      clk, rst, clken;
    input      [3:0] D;
    output reg [3:0] Q;

    always @(posedge clk)
    begin
        if (rst)
            Q <= 4'b0011;
        else if (clken)
            Q <= D;
    end
endmodule
```

ラッチ

XST で推論されるラッチには、次が含まれます。

- ・ データ入力
- ・ イネーブル入力
- ・ データ出力
- ・ セット/リセット (オプション)

ラッチの記述

ラッチは通常、ラッチ出力を記述する信号に if-else 文の分岐で新しい内容が代入されない場合に、HDL 記述から作成されます。

- ・ ラッチは次のように記述できます。
 - 同時処理信号代入文 (VHDL)

```
Q <= D when G = '1';
```

- process 文 (VHDL)

```
process (G, D)
begin
    if G = '1' then
        Q <= D;
    end process;
```

- always ブロック (Verilog)

```
always @ (G or D)
begin
    if (G)
        Q <= D;
    end
```

- ・ VHDL の場合、XST では wait 文に基づいた記述からラッチが推論されます。

ラッチの関連制約

[I/O レジスタの IOB 内へのパック](#)

ラッチのレポート

- ・ XST ログ ファイルには、認識されたラッチのタイプおよびビット幅が示されます。
- ・ 原因は、case 文や if 文が不完全であった場合など、HDL コードの間違いの結果であることがよくあります。
- ・ XST では、次のレポート例のような警告メッセージが表示されます。この警告メッセージから、推論されたラッチの機能が意図どおりであるかどうか確認できます。

ラッチのレポート例

```
=====
*                               HDL Synthesis                               *
=====

Synthesizing Unit example>.
  WARNING:Xst:737 - Found 1-bit latch for signal <Q>.
Latches may be generated from incomplete case or if statements.
We do not recommend the use of latches in FPGA/CPLD designs,
as they may lead to timing problems.

  Summary:
    inferred    1 Latch(s).
Unit example> synthesized.

=====
HDL Synthesis Report

Macro Statistics
# Latches                               : 1
  1-bit latch                           : 1

=====
```

ラッチのコード例

アップデート情報は、「はじめに」のコード例を参照してください。

ポジティブ ゲートおよび非同期リセット付きラッチの VHDL コード例

```
--
-- Latch with Positive Gate and Asynchronous Reset
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/latches/latches_2.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity latches_2 is
    port(G, D, CLR : in std_logic;
         Q : out std_logic);
end latches_2;

architecture archi of latches_2 is
begin
    process (CLR, D, G)
    begin
        if (CLR='1') then
            Q <= '0';
        elsif (G='1') then
            Q <= D;
        end if;
    end process;
end archi;
```

ポジティブ ゲート付きラッチの Verilog コード例

```
//
// Latch with Positive Gate
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/latches/latches_1.v
//
module v_latches_1 (G, D, Q);
    input G, D;
    output Q;
    reg Q;

    always @(G or D)
    begin
        if (G)
            Q = D;
    end
endmodule
```

トリステート

- ・ 通常トリステート バッファは、次で記述されます。
 - 1 つの信号
 - if - else 文
- ・ これは、バッファが次のいずれかを駆動する場合に使用されます。
 - 内部バス
 - ザイリンクス デバイスのあるボードの外部バス
- ・ 信号には、if-else の分岐の 1 つでハイ インピーダンス値を代入されます。

コード例

- ・ 同時処理信号代入文 (VHDL)

```
<= I when T = '0' else (others => 'Z');
```

- ・ 同時処理信号代入文 (Verilog)

```
assign O = (~T) ? I : 1'bZ;
```

- ・ 組み合わせプロセス文 (VHDL)

```
process (T, I)
begin
  if (T = '0') then
    O <= I;
  else
    O <= 'Z';
  end if;
end process;
```

- ・ always ブロック (Verilog)

```
always @(T or I)
begin
  if (~T)
    O = I;
  else
    O = 1'bZ;
  End
```

トリステートのインプリメンテーション

推論されたトリステート バッファは、次を駆動する際に別のデバイス プリミティブを使用してインプリメントされます。

- ・ 内部バス (BUFT)
- ・ 回路の外部ピン (OBUFT)

トリステートの関連制約

[トリステートからロジックへの変換](#)

トライステートのレポート

トライステート バッファが推論されると、HDL 合成中にレポートが出力されます。

トライステートのレポート例

```
=====
*                               HDL Synthesis                               *
=====

Synthesizing Unit example>.
  Found 1-bit tristate buffer for signal S> created at line 22
  Summary:
    inferred    8 Tristate(s).
Unit example> synthesized.

=====
HDL Synthesis Report

Macro Statistics
# Tristates                : 8
  1-bit tristate buffer    : 8
=====
```

トライステートのコード例

アップデート情報は、「はじめに」のコード例を参照してください。

組み合わせプロセスを使用したトライステートの VHDL コード例

```
--
-- Tristate Description Using Combinatorial Process
-- Implemented with an OBUFT (IO buffer)
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/tristates/tristates_1.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity three_st_1 is
    port(T : in  std_logic;
         I : in  std_logic;
         O : out std_logic);
end three_st_1;

architecture archi of three_st_1 is
begin

    process (I, T)
    begin
        if (T='0') then
            O <= I;
        else
            O <= 'Z';
        end if;
    end process;

end archi;
```

プロセス同時処理代入文を使用したトライステートの VHDL コード例

```
--  
-- Tristate Description Using Concurrent Assignment  
--  
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip  
-- File: HDL_Coding_Techniques/tristates/tristates_2.vhd  
--  
library ieee;  
use ieee.std_logic_1164.all;  
  
entity three_st_2 is  
    port(T : in  std_logic;  
          I : in  std_logic;  
          O : out std_logic);  
end three_st_2;  
  
architecture archi of three_st_2 is  
begin  
    O <= I when (T='0') else 'Z';  
end archi;
```

組み合わせプロセスを使用したトライステートの VHDL コード例

```
--
-- Tristate Description Using Combinatorial Process
-- Implemented with an OBUF (internal buffer)
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/tristates/tristates_3.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity example is

    generic (
        WIDTH : integer := 8
    );
    port(
        T : in  std_logic;
        I : in  std_logic_vector(WIDTH-1 downto 0);
        O : out std_logic_vector(WIDTH-1 downto 0));

end example;

architecture archi of example is

    signal S : std_logic_vector(WIDTH-1 downto 0);

begin

    process (I, T)
    begin
        if (T = '1') then
            S <= I;
        else
            S <= (others => 'Z');
        end if;
    end process;

    O <= not(S);

end archi;
```

組み合わせ always ブロックを使用したトライステートの Verilog コード例

```
//
// Tristate Description Using Combinatorial Always Block
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/tristates/tristates_1.v
//
module v_three_st_1 (T, I, O);
    input  T, I;
    output O;
    reg    O;

    always @(T or I)
    begin
        if (~T)
            O = I;
        else
            O = 1'bZ;
    end

endmodule
```

プロセス同時処理代入文を使用したトライステートの Verilog コード例

```
//
// Tristate Description Using Concurrent Assignment
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/tristates/tristates_2.v
//
module v_three_st_2 (T, I, O);
    input  T, I;
    output O;

    assign O = (~T) ? I: 1'bZ;

endmodule
```

カウンターおよびアキュムレーター

XST には、カウンターおよびアキュムレーターの推論機能があります。

- ・ 次のようなその他のオプションの機能を記述できます。
 - 非同期セット、リセット、ロード
 - 同期セット、リセット、ロード
 - クロック イネーブル
 - アップ、ダウン、またはアップ/ダウン方向
- ・ カウンターは、インクリメンターまたはデクリメンターとしても知られています。
- ・ XST では、符号付きおよび符号なし両方のカウンターとアキュムレーターがサポートされています。

アキュムレーターとカウンターの違い

アキュムレーターとカウンターは、加算と減算のいずれか、またはその両方でのオペランドが異なります。

カウンターの説明

- ・ デスティネーションと最初のオペランドは、信号または変数
- ・ もう 1 つ目のオペランドは、定数 1

```
A <= A + 1;
```

アキュムレーターの説明

- ・ デスティネーションと最初のオペランドは、信号または変数
- ・ 2 つ目のオペランドは次のいずれか
 - 信号または変数

```
A <= A + B;
```

- 1 以外の定数

```
A <= A + Constant;
```

推論されたカウンターまたはアキュムレーターの方向

- ・ 推論されたカウンターまたはアキュムレーターの方向：
 - **up**
 - **down**
 - **updown**
- ・ アップダウン アキュムレーターの場合、累算されるデータはアップ モードとダウン モードで別々にできます。

```
if updown = '1' then
    a <= a + b;
else
    a <= a - c;
end if;
```

ビット数

- ・ XST では、次のどちらかで記述されても、推論されたカウンタまたはアキュムレータをインプリメントするのに必要な最小のビット数が決定されます。
 - 整数型の信号
 - ビットの配列
- ・ HDL 記述で指定されていない限り、カウンタはこの数で許可された値すべてを使用します。
- ・ mod 演算子を使用すると、特定の値までカウント アップできます。次に、その例を示します。

VHDL の構文例

```
cnt <= (cnt + 1) mod MAX ;
```

Verilog の構文例

```
cnt <= (cnt + 1) %MAX;
```

カウンタおよびアキュムレータのインプリメンテーション

- ・ カウンタおよびアキュムレータは、次にインプリメントします。
 - スライス ロジック
 - DSP ブロック リソース
- ・ DSP ブロックには、最大で 2 レベルのレジスタまで吸収できます。
 - カウンタまたはアキュムレータは 1 つの DSP ブロックにフィットする必要があります。
 - カウンタまたはアキュムレータが 1 つの DSP ブロックにフィットしない場合は、スライス ロジックを使用してマクロ全体がインプリメントされます。
- ・ DSP ブロック リソースのマクロ インプリメンテーションは、[DSP ブロックの使用 \(USE_DSP48\)](#) 制約をデフォルト値の auto に設定して制御します。
- ・ auto に設定すると、XST では次を考慮してカウンタおよびアキュムレータがインプリメントされます。
 - デバイスで使用可能な DSP ブロック リソース
 - 累積されたデータ ソースのようなコンテキスト情報
 - DSP ブロックへのインプリメンテーションでハイ パフォーマンスのザイリンクス DSP ブロックのカスケード機能が使用できるかどうか
- ・ ほとんどのスタンドアロンのカウンタおよびアキュムレータで、スライス ロジックはデフォルトの auto モードを使用することをお勧めしますが、DSP ブロックにインプリメンテーションを強制する場合は、yes に変更してください。
- ・ また、auto モードの場合は、[DSP 使用率 \(DSP_UTILIZATION_RATIO\)](#) 制約で DSP48 リソースの使用が制御されます。XST では、使用可能な DSP ブロック リソースをすべて使用しようとします。
- ・ 詳細は、「[四則演算の DSP ブロック リソース](#)」を参照してください。

カウンターおよびアキュムレーターの関連制約

- ・ [DSP ブロックの使用](#)
- ・ [DSP 使用率](#)

カウンターおよびアキュムレーターのレポート

カウンターおよびアキュムレーターは、次の組み合わせにしたがってアドバンス HDL 合成中に認識されます。

- ・ レジスタ、および
- ・ HDL 合成中に推論された加算器/減算器マクロ

カウンターおよびアキュムレーターのレポートの例

```
=====
*                               HDL Synthesis                               *
=====

Synthesizing Unit <example>.
  Found 4-bit register for signal <cnt>.
  Found 4-bit register for signal <acc>.
  Found 4-bit adder for signal <n0005> created at line 29.
  Found 4-bit adder for signal <n0006> created at line 30.
  Summary:
  inferred   2 Adder/Subtractor(s).
  inferred   8 D-type flip-flop(s).
Unit <example> synthesized.

=====
HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                : 2
  4-bit adder                       : 2
# Registers                         : 2
  4-bit register                    : 2

=====

=====
*                               Advanced HDL Synthesis                       *
=====

Synthesizing (advanced) Unit <example>.
The following registers are absorbed into counter <cnt>: 1 register on signal <cnt>.
The following registers are absorbed into accumulator <acc>: 1 register on signal <acc>.
Unit <example> synthesized (advanced).

=====
Advanced HDL Synthesis Report

Macro Statistics
# Counters                          : 1
  4-bit up counter                  : 1
# Accumulators                      : 1
  4-bit up accumulator              : 1

=====
```

カウンターおよびアキュムレーターのコード例

アップデート情報は、「はじめに」のコード例を参照してください。

同期リセット付き 4 ビット符号なしアップ アキュムレーターの VHDL コード例

```
--
-- 4-bit Unsigned Up Accumulator with synchronous Reset
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/accumulators/accumulators_2.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity accumulators_2 is
    generic (
        WIDTH : integer := 4);
    port (
        clk : in  std_logic;
        rst : in  std_logic;
        D   : in  std_logic_vector(WIDTH-1 downto 0);
        Q   : out std_logic_vector(WIDTH-1 downto 0));
end accumulators_2;

architecture archi of accumulators_2 is
    signal cnt : std_logic_vector(WIDTH-1 downto 0);
begin

    process (clk)
    begin
        if rising_edge(clk) then
            if (rst = '1') then
                cnt <= (others => '0');
            else
                cnt <= cnt + D;
            end if;
        end if;
    end process;

    Q <= cnt;

end archi;
```

同期ロード付き 4 ビット符号なしダウン カウンターの Verilog コード例

```
//  
// 4-bit unsigned down counter with a synchronous load.  
//  
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip  
// File: HDL_Coding_Techniques/counters/counters_31.v  
//  
module v_counters_31 (clk, load, Q);  
  
    parameter WIDTH = 4;  
    input  clk;  
    input  load;  
    output [WIDTH-1:0] Q;  
    reg    [WIDTH-1:0] cnt;  
  
    always @(posedge clk)  
    begin  
        if (load)  
            cnt <= {WIDTH{1'b1}};  
        else  
            cnt <= cnt - 1'b1;  
        end  
  
        assign Q = cnt;  
  
endmodule
```

シフトレジスタ

シフトレジスタは、フリップフロップのチェーンで、これにより決まった数 (スタティック) のレイテンシ ステージをまたいでデータを伝搬できます。 [ダイナミック シフトレジスタ](#) では、伝搬チェーンの長さが回路の操作中にダイナミックに変更されます。

スタティック シフトレジスタ エlement

スタティック シフトレジスタには、通常次の含まれます。

- ・ クロック
- ・ クロック イネーブル (オプション)
- ・ シリアル データ入力
- ・ シリアル データ出力

その他の機能の含有

- ・ リセットやセット、パラレル ロード ロジックなどを追加で含めることができますが、
- ・ 追加で機能を含めると、SRL タイプの専用プリミティブを利用して、デバイス使用率を削減して最適なパフォーマンスを実現することができないことがあります。
- ・ ザイリンクスではこのようなロジックを削除して、内容をシリアルに読み込むことをお勧めしています。

シフトレジスタの記述

次のコード例では、シフトレジスタのコア ファンクションを記述する 2 つの方法を示しています。

連結演算子を使用した VHDL コード例

次のコード例では、連結演算子を使用してシフトレジスタのコア ファンクションをコンパクトに記述しています。

```
shreg <= shreg (6 downto 0) & SI;
```

for-loop 文の VHDL コード例

次のコード例では、for loop 文を使用してシフトレジスタのコア ファンクションを記述しています。

```
for i in 0 to 6 loop
    shreg(i+1) <= shreg(i);
end loop;
shreg(0) <= SI;
```

シフトレジスタのインプリメンテーション

シフトレジスタのインプリメンテーションには、次が含まれます。

- ・ シフトレジスタの SRL ベースのインプリメンテーション
- ・ ブロック RAM へのシフトレジスタのインプリメンテーション
- ・ LUT RAM へのシフトレジスタのインプリメンテーション

シフトレジスタの SRL ベースのインプリメンテーション

- ・ 推論されたシフトレジスタは、SRL タイプのリソースにインプリメントされます。
 - SRL16
 - SRL16E
 - SRLC16
 - SRLC16E
 - SRLC32E
- ・ シフトレジスタの長さによって、XST では次のいずれかが実行されます。
 - 1 つの SRL タイプのプリミティブにインプリメント
 - SRLC タイプのプリミティブのカスケード機能を使用
- ・ また、残りのデザインでシフトレジスタの中間地点のどこかが使用される場合も、このカスケード機能が使用されます。
- ・ 遅延線は、SRL タイプのリソースではなく、RAM リソース (ブロック RAM、LUT RAM) にインプリメントできます。RAM リソースに遅延線をインプリメントすると、遅延線が長くなった場合に、電力が保存されやすくなるという利点があります。
- ・ 「シフトレジスタの記述」で説明するようにブロック RAM または LUT RAM にはシフトレジスタをインプリメントできません。RAM ベースのインプリメンテーションは次のコード例のように明確に記述する必要があります。

ブロック RAM へのシフトレジスタのインプリメンテーション

- ・ read-first 同期モードには、次のようなカウンターが含まれます。
 - アドレス指定可能な空間を順番にスキャン
 - 遅延線の長さが -2 に到達したときに 0 までカウント バック
- ・ 最大のパフォーマンスを得るには、ブロック RAM の出力ラッチとオプションの出力レジスタステージを使用します。次に例を示します。
 - 深さ 512 の遅延線の場合、RAM のアドレス指定可能なデータワード 510 が使用されます。
 - データ出力ラッチとオプションの出力レジスタが最後の 2 ステージを提供します。
- ・ 詳細は、「RAM の HDL コード記述のガイドライン」を参照してください。

ブロック RAM に深さ 512、8 ビットの遅延線をインプリメントする VHDL コード例

```
--  
-- A 512-deep 8-bit delay line implemented on block RAM  
-- 510 stages implemented as addressable memory words  
-- 2 stages implemented with output latch and optional output register for  
-- optimal performance
```

```

--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/shift_registers/delayline_bram_512.vhd
--

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity srl_512_bram is
  generic (
    LENGTH      : integer := 512;
    ADDRWIDTH   : integer := 9;
    WIDTH       : integer := 8);
  port (
    CLK         : in  std_logic;
    SHIFT_IN    : in  std_logic_vector(WIDTH-1 downto 0);
    SHIFT_OUT   : out std_logic_vector(WIDTH-1 downto 0));
end srl_512_bram;

architecture behavioral of srl_512_bram is

  signal CNTR : std_logic_vector(ADDRWIDTH-1 downto 0);
  signal SHIFT_TMP : std_logic_vector(WIDTH-1 downto 0);
  type ram_type is array (0 to LENGTH-3) of std_logic_vector(WIDTH-1 downto 0);
  signal RAM : ram_type := (others => (others => '0'));
begin

  counter : process (CLK)
  begin
    if CLK'event and CLK = '1' then
      if CNTR = conv_std_logic_vector(LENGTH-3, ADDRWIDTH) then
        CNTR <= (others => '0');
      else
        CNTR <= CNTR + '1';
      end if;
    end if;
  end process counter;

  memory : process (CLK)
  begin
    if CLK'event and CLK = '1' then
      RAM(conv_integer(CNTR)) <= SHIFT_IN;
      SHIFT_TMP <= RAM(conv_integer(CNTR));
      SHIFT_OUT <= SHIFT_TMP;
    end if;
  end process memory;

```

```
end behavioral;
```

ブロック RAM に深さ 514、8 ビットの遅延線をインプリメントする VHDL コード例

```
--
-- A 514-deep 8-bit delay line implemented on block RAM
-- 512 stages implemented as addressable memory words
-- 2 stages implemented with output latch and optional output register for
-- optimal performance
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/shift_registers/delayline_bram_514.vhd
--
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity srl_514_bram is
  generic (
    LENGTH      : integer := 514;
    ADDRWIDTH   : integer := 9;
    WIDTH       : integer := 8);
  port (
    CLK         : in  std_logic;
    SHIFT_IN    : in  std_logic_vector(WIDTH-1 downto 0);
    SHIFT_OUT   : out std_logic_vector(WIDTH-1 downto 0));
end srl_514_bram;

architecture behavioral of srl_514_bram is

  signal CNTR : std_logic_vector(ADDRWIDTH-1 downto 0);
  signal SHIFT_TMP : std_logic_vector(WIDTH-1 downto 0);
  type ram_type is array (0 to LENGTH-3) of std_logic_vector(WIDTH-1 downto 0);
  signal RAM : ram_type := (others => (others => '0'));
begin

  counter : process (CLK)
  begin
    if CLK'event and CLK = '1' then
      CNTR <= CNTR + '1';
    end if;
  end process counter;

  memory : process (CLK)
  begin
    if CLK'event and CLK = '1' then
```

```

    RAM(conv_integer(CNTR)) <= SHIFT_IN;
    SHIFT_TMP                <= RAM(conv_integer(CNTR));
    SHIFT_OUT                <= SHIFT_TMP;
  end if;
end process memory;

end behavioral;

```

LUT RAM へのシフトレジスタのインプリメンテーション

- ・ シフトレジスタは分散 RAM にインプリメントできます。
- ・ 最後のステージは別のレジスタを使用してインプリメントされます。たとえば、深さ 128 の遅延線では次が使用されます。
 - 127 ワードのアドレス指定可能なデータを含む LUT RAM
 - 最後のレジスタ ステージ
- ・ 詳細は、「RAM の HDL コード記述のガイドライン」を参照してください。

LUT RAM に深さ 128、8 ビットの遅延線をインプリメントする VHDL コード例

```

--
-- A 128-deep 8-bit delay line implemented on LUT RAM
-- 127 stages implemented as addressable memory words
-- Last stage implemented with an external register
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/shift_registers/delayline_lutram_128.vhd
--
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity srl_128_lutram is
  generic (
    LENGTH      : integer := 128;
    ADDRWIDTH   : integer := 7;
    WIDTH       : integer := 8);
  port (
    CLK         : in  std_logic;
    SHIFT_IN    : in  std_logic_vector(WIDTH-1 downto 0);
    SHIFT_OUT   : out std_logic_vector(WIDTH-1 downto 0));
end srl_128_lutram;

architecture behavioral of srl_128_lutram is

  signal CNTR : std_logic_vector(ADDRWIDTH-1 downto 0);
  type ram_type is array (0 to LENGTH-2) of std_logic_vector(WIDTH-1 downto 0);

```



```
signal RAM : ram_type := (others => (others => '0'));

attribute ram_style : string;
attribute ram_style of RAM : signal is "distributed";

begin

counter : process (CLK)
begin
    if CLK'event and CLK = '1' then
        if CNTR = conv_std_logic_vector(LENGTH-2, ADDRWIDTH) then
            CNTR <= (others => '0');
        else
            CNTR <= CNTR + '1';
        end if;
    end if;
end process counter;

memory : process (CLK)
begin
    if CLK'event and CLK = '1' then
        RAM(conv_integer(CNTR)) <= SHIFT_IN;
        SHIFT_OUT <= RAM(conv_integer(CNTR));
    end if;
end process memory;

end behavioral;
```

シフトレジスタの関連制約

シフトレジスタ抽出

シフトレジスタのレポート

HDL 合成中、XST は最初に個々のフリップフロップを識別します。実際にシフトレジスタが認識されるのは、下位レベルの合成中です。

シフトレジスタのレポート例

```
=====
* HDL Synthesis *
=====
Synthesizing Unit <example>.
    Found 8-bit register for signal <tmp>.
    Summary:
        inferred 8 D-type flip-flop(s).
Unit <example> synthesized.

(...)

=====
* Advanced HDL Synthesis *
=====
Advanced HDL Synthesis Report
Macro Statistics
# Registers : 8
Flip-Flops : 8
=====

(...)

=====
* Low Level Synthesis *
=====
Processing Unit <example> :
    Found 8-bit shift register for signal <tmp_7>.
Unit <example> processed.

(...)

=====
Final Register Report
Macro Statistics
# Shift Registers : 1
8-bit shift register : 1
=====
```

シフト レジスタのコード例

アップデート情報は、「はじめに」のコード例を参照してください。

32 ビットのシフト レジスタの VHDL コード例 1

次のコード例では、連結コード スタイルを使用しています。

```
--
-- 32-bit Shift Register
--     Rising edge clock
--     Active high clock enable
--     Concatenation-based template
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/shift_registers/shift_registers_0.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_0 is

    generic (
        DEPTH : integer := 32
    );
    port (
        clk    : in  std_logic;
        clken  : in  std_logic;
        SI     : in  std_logic;
        SO     : out std_logic);

end shift_registers_0;

architecture archi of shift_registers_0 is
    signal shreg: std_logic_vector(DEPTH-1 downto 0);
begin

    process (clk)
    begin
        if rising_edge(clk) then
            if clken = '1' then
                shreg <= shreg(DEPTH-2 downto 0) & SI;
            end if;
        end if;
    end process;

    SO <= shreg(DEPTH-1);

end archi;
```

32 ビットのシフトレジスタの VHDL コード例 2

同じ機能は、次のように記述できます。

```
--
-- 32-bit Shift Register
--     Rising edge clock
--     Active high clock enable
--     for loop-based template
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/shift_registers/shift_registers_1.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_1 is

    generic (
        DEPTH : integer := 32
    );
    port (
        clk    : in  std_logic;
        clken   : in  std_logic;
        SI      : in  std_logic;
        SO      : out std_logic);

end shift_registers_1;

architecture archi of shift_registers_1 is
    signal shreg: std_logic_vector(DEPTH-1 downto 0);
begin

    process (clk)
    begin
        if rising_edge(clk) then
            if clken = '1' then
                for i in 0 to DEPTH-2 loop
                    shreg(i+1) <= shreg(i);
                end loop;
                shreg(0) <= SI;
            end if;
        end if;
    end process;

    SO <= shreg(DEPTH-1);

end archi;
```

8 ビットのシフトレジスタの Verilog コード例 1

次のコード例では、連結コードスタイルを使用してレジスタチェーンを記述しています。

```
//  
// 8-bit Shift Register  
//   Rising edge clock  
//   Active high clock enable  
//   Concatenation-based template  
//  
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip  
// File: HDL_Coding_Techniques/shift_registers/shift_registers_0.v  
//  
module v_shift_registers_0 (clk, clken, SI, SO);  
  
    parameter WIDTH = 8;  
    input  clk, clken, SI;  
    output SO;  
    reg    [WIDTH-1:0] shreg;  
  
    always @(posedge clk)  
    begin  
        if (clken)  
            shreg = {shreg[WIDTH-2:0], SI};  
    end  
  
    assign SO = shreg[WIDTH-1];  
  
endmodule
```

8 ビットのシフトレジスタの Verilog コード例 2

```
//
// 8-bit Shift Register
//      Rising edge clock
//      Active high clock enable
//      For-loop based template
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/shift_registers/shift_registers_1.v
//
module v_shift_registers_1 (clk, clken, SI, SO);

    parameter WIDTH = 8;
    input  clk, clken, SI;
    output SO;
    reg    [WIDTH-1:0] shreg;

    integer i;

    always @(posedge clk)
    begin
        if (clken)
        begin
            for (i = 0; i < WIDTH-1; i = i+1)
                shreg[i+1] <= shreg[i];
            shreg[0] <= SI;
        end
    end

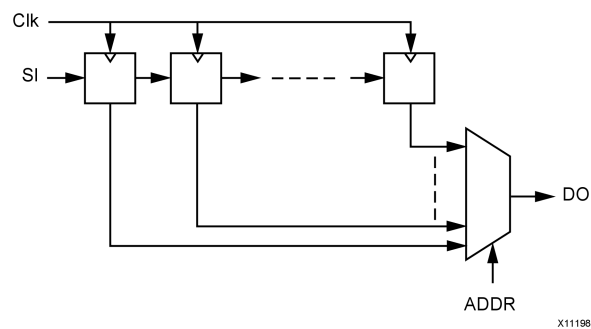
    assign SO = shreg[WIDTH-1];

endmodule
```

ダイナミック シフト レジスタ

- ・ ダイナミック シフト レジスタは、回路操作中にダイナミックに長さを変えることができるシフトレジスタです。
- ・ ダイナミック シフト レジスタは、次のように認識できます。
 - 回路操作中に使用される最大長のフリップフロップのチェーン
 - 指定されたクロック サイクルでデータがその伝搬チェーンから抽出されるステージを選択するマルチプレクサ
- ・ XST では、どのような最大長のダイナミック シフト レジスタでも推論できます。
- ・ XST では、デバイス ファミリーで使用可能な SRL タイプのプリミティブを使用してダイナミック シフト レジスタを最適にインプリメントできます。

ダイナミック シフト レジスタの図



ダイナミック シフト レジスタの関連制約

シフトレジスタ抽出

ダイナミック シフト レジスタのレポート

- ・ HDL 合成中、XST はフリップフロップとマルチプレクサを識別します。
- ・ アドバンス HDL 合成中は次が実行されます。
 - ダイナミック シフト レジスタを識別
 - フリップフロップとマルチプレクサ間の依存関係を決定

ダイナミック シフト レジスタのレポート例

```

=====
* HDL Synthesis *
=====

Synthesizing Unit <example>.
    Found 1-bit 16-to-1 multiplexer for signal <Q>.
    Found 16-bit register for signal <SRL_SIG>.
    Summary:
        inferred 16 D-type flip-flop(s).
        inferred 1 Multiplexer(s).
Unit <example> synthesized.

(...)
=====
* Advanced HDL Synthesis *
=====

Synthesizing (advanced) Unit <example>.
    Found 16-bit dynamic shift register for signal <Q>.
Unit <example> synthesized (advanced).

=====
HDL Synthesis Report
Macro Statistics
# Shift Registers : 1
16-bit dynamic shift register : 1
=====

```

ダイナミック シフト レジスタのコード例

アップデート情報は、「はじめに」のコード例を参照してください。

32 ビットのダイナミック シフト レジスタの VHDL コード例

```

--
-- 32-bit dynamic shift register.
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/dynamic_shift_registers/dynamic_shift_registers_1.vhd
--
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity example is

    generic (
        DEPTH      : integer := 32;
        SEL_WIDTH  : integer := 5
    )

```



```
);
port(
    CLK : in  std_logic;
    SI   : in  std_logic;
    CE   : in  std_logic;
    A    : in  std_logic_vector(SEL_WIDTH-1 downto 0);
    DO   : out std_logic
);

end example;

architecture rtl of example is

    type SRL_ARRAY is array (0 to DEPTH-1) of std_logic;
    -- The type SRL_ARRAY can be array
    -- (0 to DEPTH-1) of
    -- std_logic_vector(BUS_WIDTH downto 0)
    -- or array (DEPTH-1 downto 0) of
    -- std_logic_vector(BUS_WIDTH downto 0)
    -- (the subtype is forward (see below))
    signal SRL_SIG : SRL_ARRAY;

begin
    process (CLK)
    begin
        if rising_edge(CLK) then
            if CE = '1' then
                SRL_SIG <= SI & SRL_SIG(0 to DEPTH-2);
            end if;
        end if;
    end process;

    DO <= SRL_SIG(conv_integer(A));

end rtl;
```

32 ビットのダイナミック シフト レジスタの Verilog コード例

```
//  
// 32-bit dynamic shift register.  
//  
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip  
// File: HDL_Coding_Techniques/dynamic_shift_registers/dynamic_shift_registers_1.v  
//  
module v_dynamic_shift_registers_1 (CLK, CE, SEL, SI, DO);  
  
    parameter SELWIDTH = 5;  
    input      CLK, CE, SI;  
    input      [SELWIDTH-1:0] SEL;  
    output     DO;  
  
    localparam DATAWIDTH = 2**SELWIDTH;  
    reg [DATAWIDTH-1:0] data;  
  
    assign DO = data[SEL];  
  
    always @(posedge CLK)  
    begin  
        if (CE == 1'b1)  
            data <= {data[DATAWIDTH-2:0], SI};  
    end  
  
endmodule
```

マルチプレクサ

- ・ マルチプレクサ マクロは、次のようなさまざまなコード スタイルから推論できます。
 - 同時処理代入文
 - 組み合わせプロセスまたは always ブロックで記述
 - 順次プロセスまたは always ブロックで記述
- ・ マルチプレクサの記述には、通常次が含まれます。
 - if-elsif 文
 - case 文
- ・ case 文を使用する場合は、次に注意してください。
 - セレクタ値をすべて列挙
 - デフォルト文でどのデータが明示的に列挙されていないセレクタ値に対して選択されているかを定義
- ・ これらをしておかないと、不必要なラッチが作成されてしまいます。マルチプレクサが if-elsif 文で記述される場合に、else 文がない場合も、不必要なラッチが作成されます。
- ・ 同じデータがセレクタの別の値に対して選択される場合、don't care を使用してこれらのセレクタ値をコンパクトな方法で記述できます。

マルチプレクサのインプリメンテーション

マルチプレクサ マクロを明示的に推論するかどうかは、マルチプレクサの入力、特に共通する入力の数によって異なります。

マルチプレクサの Verilog の [Case Implementation Style] パラメーター

- ・ [Case Implementation Style] パラメーターを使用すると、case 文をさらに指定することができます。
- ・ [Case Implementation Style] を full、parallel、または full-parallel に設定すると、最初のモデルのビヘイビアーと異なるビヘイビアーがインプリメントされる場合があります。
- ・ 詳細は、第 9 章「[デザイン制約](#)」を参照してください。

case 文のインプリメンテーション スタイル パラメーターの値

- ・ none (デフォルト)
case 文のビヘイビアーを記述どおりにインプリメントします。
- ・ full
 - XST では、case 文が完了していると認識されます。
 - 可能性のあるセレクトの値すべてが列挙されていなかったとしても、ラッチが作成されないようにします。
- ・ parallel
 - XST では分岐は同時に発生できないと判断されます。
 - プライオリティ エンコーディング ロジックは作成されません。
- ・ full-parallel
 - case 文が完了し、分岐は同時に発生できないと判断されます。
 - ラッチおよびプライオリティ エンコーディング ロジックは作成されません。

XST のメッセージ

- ・ [Case Implementation Style] パラメーターが実際に使用されると、メッセージが表示されます。
- ・ case 文の特性からその必要がない場合、メッセージは表示されません。たとえば case 文でセレクトに使用される可能性のある値すべてを列挙するようになっている場合に case パラメーターを full にした場合、メッセージは必要ありません。

マルチプレクサーの関連制約

列挙型エンコード手法

マルチプレクサのレポート

- ・ XST ログ ファイルには、認識されたマルチプレクサのタイプおよびビット幅が示されます。
- ・ マルチプレクサの明示的推論およびレポートは、マルチプレクサのサイズによって異なります。たとえば、4:1 マルチプレクサはレポートされません。マルチプレクサが 8:1 以上のサイズであると推論されます。

マルチプレクサのレポート例

```
=====
*                               HDL Synthesis                               *
=====
```

Synthesizing Unit <example>.

Found 1-bit 8-to-1 multiplexer for signal <o> created at line 11.

Summary:

inferred 1 Multiplexer(s).

Unit <example> synthesized.

```
=====
HDL Synthesis Report
```

Macro Statistics

# Multiplexers	: 1
1-bit 8-to-1 multiplexer	: 1

```
=====
```

マルチプレクサーのコード例

アップデート情報は、「はじめに」のコード例を参照してください。

if 文を使用した 8:1 の 1 ビット MUX の VHDL コード例

```
//
// 8-to-1 1-bit MUX using an If statement.
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/multiplexers/multiplexers_1.v
//
module v_multiplexers_1 (di, sel, do);
    input [7:0] di;
    input [2:0] sel;
    output reg do;

    always @(sel or di)
    begin
        if      (sel == 3'b000) do = di[7];
        else if (sel == 3'b001) do = di[6];
        else if (sel == 3'b010) do = di[5];
        else if (sel == 3'b011) do = di[4];
        else if (sel == 3'b100) do = di[3];
        else if (sel == 3'b101) do = di[2];
        else if (sel == 3'b110) do = di[1];
        else
            do = di[0];
    end
endmodule
```

if 文を使用した 8:1 の 1 ビット MUX の Verilog コード例

```
//
// 8-to-1 1-bit MUX using an If statement.
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/multiplexers/multiplexers_1.v
//
module v_multiplexers_1 (di, sel, do);
    input [7:0] di;
    input [2:0] sel;
    output reg do;

    always @(sel or di)
    begin
        if      (sel == 3'b000) do = di[7];
        else if (sel == 3'b001) do = di[6];
        else if (sel == 3'b010) do = di[5];
        else if (sel == 3'b011) do = di[4];
        else if (sel == 3'b100) do = di[3];
        else if (sel == 3'b101) do = di[2];
        else if (sel == 3'b110) do = di[1];
        else
            do = di[0];
    end
endmodule
```

case 文を使用した 8:1 の 1 ビット MUX の VHDL コード例

```
--
-- 8-to-1 1-bit MUX using a Case statement.
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/multiplexers/multiplexers_2.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity multiplexers_2 is

    port (di   : in  std_logic_vector(7 downto 0);
          sel  : in  std_logic_vector(2 downto 0);
          do   : out std_logic);

end multiplexers_2;

architecture archi of multiplexers_2 is
begin
    process (sel, di)
    begin
        case sel is
            when "000" => do <= di(7);
            when "001" => do <= di(6);
            when "010" => do <= di(5);
            when "011" => do <= di(4);
            when "100" => do <= di(3);
            when "101" => do <= di(2);
            when "110" => do <= di(1);
            when others => do <= di(0);
        end case;
    end process;
end archi;
```


case 文を使用した 8:1 の 1 ビット MUX の Verilog コード例

```
//
// 8-to-1 1-bit MUX using a Case statement.
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/multiplexers/multiplexers_2.v
//
module v_multiplexers_2 (di, sel, do);
    input [7:0] di;
    input [2:0] sel;
    output reg do;

    always @(sel or di)
    begin
        case (sel)
            3'b000 : do = di[7];
            3'b001 : do = di[6];
            3'b010 : do = di[5];
            3'b011 : do = di[4];
            3'b100 : do = di[3];
            3'b101 : do = di[2];
            3'b110 : do = di[1];
            default : do = di[0];
        endcase
    end
endmodule
```

トリステート バッファを使用した 8:1 の 1 ビット MUX の Verilog コード例

```
//
// 8-to-1 1-bit MUX using tristate buffers.
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/multiplexers/multiplexers_3.v
//
module v_multiplexers_3 (di, sel, do);
    input [7:0] di;
    input [7:0] sel;
    output      do;

    assign do = sel[0] ? di[0] : 1'bz;
    assign do = sel[1] ? di[1] : 1'bz;
    assign do = sel[2] ? di[2] : 1'bz;
    assign do = sel[3] ? di[3] : 1'bz;
    assign do = sel[4] ? di[4] : 1'bz;
    assign do = sel[5] ? di[5] : 1'bz;
    assign do = sel[6] ? di[6] : 1'bz;
    assign do = sel[7] ? di[7] : 1'bz;

endmodule
```

四則演算ブロックの HDL コーディング手法

XST では、基本的な四則演算がサポートされています。

- ・ 加算器、減算器、加減算器
- ・ 乗算器
- ・ 除算器
- ・ コンパレータ

これらの基本的な演算マクロからは、次のようなより複雑なブロックを構築できます。

- ・ アキュムレーター
詳細は、「[カウンタおよびアキュムレーター](#)」を参照してください。
- ・ 乗算/加算器
詳細は、「[乗加算と乗累算](#)」を参照してください。
- ・ DSP フィルター
詳細は、「[四則演算の DSP ブロック リソース](#)」を参照してください。

四則演算の符号サポート

- ・ XST では、次の符号付きおよび符号なしの四則演算がサポートされています。
 - 加算器
 - 減算器
 - コンパレータ
 - 乗算器
- ・ Verilog または VHDL では、加算器やカウンタなどの一部のマクロは、符号付きおよび符号なしの両方の値用にインプリメントできます。

Verilog の符号サポート

- ・ 明示的に記述しなくても、Verilog では次の表に示す規則が適用されます。
- ・ データ型の記述を明示的に指定するには、signed と unsigned キーワードを使用します。

符号付き、符号なしの使用規則

コンポーネント	符号	例外
port、wire、reg ベクター タイプ	unsigned	signed と明確に宣言されない場合
整数変数	signed	指定されない限り
10 進数	signed	なし
基数	unsigned	指定されない限り

論理式タイプの定義

- ・ 論理式のタイプ :
 - オペランドでのみ定義されます。
 - 代入文の左側部分のタイプには関係ありません。
- ・ 自ら決定されるオペランドの符号とビット長 :
 - オペランド自体で決定されます。
 - 残りの論理式とは関係ありません。
- ・ コンテキストで決定されるオペランドを使用する場合は、Verilog LRM のガイドラインを参照してください。

論理式タイプの決定

論理式タイプは、次の表の規則に従って決定されます。

論理式タイプの決定規則

結果	ステータス	オペランド
ビット セレクト	符号なし	オペランドに関係なし
パーツ セレクト	符号なし	パーツ セレクトでベクター全体が指定されていたとしても、オペランドに関係なく符号なし
連結	符号なし	オペランドに関係なし
比較	符号なし	オペランドに関係なし

Verilog の符号サポートのコード例 1

```
input signed    [31:0] example1;
reg unsigned    [15:0] example2;
wire signed     [31:0] example3;
```

Verilog の符号サポートのコード例 2

基数指定子に s を使用すると、基数を符号付きに指定できます。

```
4'sd87
```

Verilog の符号サポートのコード例 3

\$signed および \$unsigned 変換関数を使用すると、符号付きまたは符号なしに指定できます。

```
wire [7:0] udata;
wire [7:0] sdata;
assign sdata = $signed(udata);
```

VHDL の符号サポート

- ・ VHDL では、オペランドの演算およびタイプによって、コードに追加のパッケージを含める必要があります。
- ・ 使用可能なタイプの詳細については、IEEE 規格の VHDL のマニュアルを参照してください。

符号なし加算器

符号なし加算器を作成するには、次の表のような符号なしの値を処理する演算パッケージおよびタイプを使用します。

パッケージ	タイプ
numeric_std	unsigned
std_logic_arith	unsigned
std_logic_unsigned	std_logic_vector

符号付き加算器

符号付き加算器を作成するには、次の表のような符号付きの値 (signed) を処理する演算パッケージおよびタイプを使用します。

パッケージ	タイプ
numeric_std	signed
std_logic_arith	signed
std_logic_signed	std_logic_vector

四則演算のインプリメンテーション

四則演算のインプリメンテーションには、次が含まれます。

- ・ [四則演算のスライス ロジック](#)
- ・ [四則演算の DSP ブロック リソース](#)

四則演算のスライス ロジック

四則演算マクロをスライス ロジックにインプリメントする場合、XST では高速で効率的な演算ファンクションをインプリメントするための専用キャリー ロジックといった、ザイリンクスの CLB 構造の機能が利用されます。

四則演算の DSP ブロック リソース

Virtex®-6、Spartan®-6 および 7 シリーズ デバイスには、専用の高パフォーマンス演算ブロック (DSP ブロック) が含まれます。

- ・ DSP ブロックの数は、デバイスによって異なります。
 - ・ DSP ブロックのコンフィギュレーションを変更して、さまざまな演算ファンクションをインプリメントできます。
 - ・ 可能性が最大限に引き出された場合、DSP ブロックは完全にパイプライン化された preadder-multiply-add (前置加算器 - 乗算 - 加算) または preadder-multiply-accumulate (前置加算器 - 乗算 - 累算) ファンクションをインプリメントできます。
 - ・ XST では、これらのリソースを利用して、高パフォーマンスで電力効率の良い演算ロジックをインプリメントします。
 - ・ 演算マクロをスライス ロジックにインプリメントするか DSP ブロック リソースにインプリメントするかは、[DSP ブロックの使用 \(USE_DSP48\)](#) 制約を auto に設定して制御します。
 - ・ auto モードの場合、XST では実際に使用可能な DSP ブロック リソースが考慮され、デバイスがオーバーマップされないようになります。
 - XST では、使用可能な DSP ブロック リソースがすべて使用される可能性があります。
 - [DSP 使用率 \(DSP_UTILIZATION_RATIO\)](#) を使用すると、これらのリソースを割り当てないままの状態にしておくことができます。
 - ・ 演算マクロの中には、デフォルトでは DSP ブロックにインプリメントされないものがあります。
 - これらのマクロを強制的にインプリメントするには、[DSP ブロックの使用 \(USE_DSP48\)](#) 制約の値を yes にします。
 - これらは、スタンドアロン次のマクロです。
 - ◆ 加算器
 - ◆ アキュムレーター
 - ◆ カウンター
 - ・ DSP ブロックにインプリメントされる演算ファンクションをパイプライン化する場合に、その利点が活かされるようにするには、レジスタにオプションでクロック イネーブルを付けて記述します。
 - レジスタは、オプションで同時にリセットできます。
 - 非同期リセット ロジックを使用すると、このようなインプリメンテーションを回避できます。
 - ザイリンクスでは非同期リセット ロジックの使用はお勧めしていません。
 - ・ DSP ブロック リソースでは、オペランドが符号付きとみなされます。符号なし (unsigned) の演算を記述する場合、符号なしのオペランドを 1 つの DSP ブロックの全幅にマップすることはできません。
- 例
- XST は 1 つの Virtex-6 DSP48E1 ブロックに最大で 25 X 18 ビットの符号付き (signed) 乗算をインプリメントできます。
 - DSP ブロック入力の MSB (最上位ビット) を 0 にすると、同じ 1 つのブロックにのみ最大 24 X 17 ビットの符号なしの製品をインプリメントできます。

詳細情報

- ・ DSP ブロック リソースのインプリメンテーションについては、次を参照してください。
 - [乗算器](#)
 - [乗加算および乗累算](#)
- ・ DSP ブロック リソースの詳細は、次を参照してください。
 - [『Virtex-6 FPGA DSP48E1 スライス ユーザー ガイド』\(UG369\) \(ザイリンクス サポート ウェブサイトより\)](#)
 - [『Spartan-6FPGA DSP48A1 スライス ユーザー ガイド』\(UG389\) \(ザイリンクス サポート ウェブサイトより\)](#)

コンパレータ

XST では、次のすべてのタイプのコンパレータが認識されます。

- ・ 等価
- ・ 非等価
- ・ 大なり
- ・ 大なり、または等価
- ・ より小さい
- ・ 小なり、または等価

コンパレータの関連制約

なし

コンパレータのレポート

- ・ 信号または変数の定数に対する equal (=) または not equal (≠) は、XST で直接ブール型ロジックに最適化されるので、明示的なコンパレータ マクロ推論にはなりません。
- ・ その他すべての比較では、コンパレータ マクロの推論は次のようにレポートされます。

コンパレータのレポート例

```
=====
*                               HDL Synthesis                               *
=====

Synthesizing Unit <example>.
  Found 8-bit comparator lessequal for signal <n0000> created at line 8
  Found 8-bit comparator greater for signal <cmp2> created at line 15
  Summary:
  inferred   2 Comparator(s).
Unit <example> synthesized.

=====
HDL Synthesis Report

Macro Statistics
# Comparators                : 2
 8-bit comparator greater     : 1
 8-bit comparator lessequal   : 1
=====
```

コンパレータのコード例

アップデート情報は、「はじめに」のコード例を参照してください。

符号なし 8 ビット コンパレータの VHDL コード例

```
--
-- Unsigned 8-bit Greater or Equal Comparator
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/comparators/comparators_1.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity comparators_1 is
    generic (
        WIDTH : integer := 8);
    port (
        A,B : in  std_logic_vector(WIDTH-1 downto 0);
        CMP : out std_logic);
end comparators_1;

architecture archi of comparators_1 is
begin
    CMP <= '1' when A >= B else '0';
end archi;
```

符号なし 8 ビットの小さなコンパレータの Verilog コード例

```
//
// Unsigned 8-bit Less Than Comparator
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/comparators/comparators_1.v
//
module v_comparators_1 (A, B, CMP);

    parameter WIDTH = 8;
    input  [WIDTH-1:0] A;
    input  [WIDTH-1:0] B;
    output CMP;

    assign CMP = (A < B) ? 1'b1 : 1'b0;

endmodule
```

除算器

XST では、次の場合のみ除算器がサポートされます。

- ・ 除数が定数および 2 のべき乗の場合 (この記述は、シフターとしてインプリメントされます)
- ・ どちらのオペランドも定数の場合

その他の場合、XST はエラー メッセージを表示して終了します。

除算器の関連制約

なし

除算器のレポート

なし

除算器のコード例

アップデート情報は、「はじめに」のコード例を参照してください。

定数 2 で除算する VHDL コード例

```
--
-- Division By Constant 2
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/dividers/dividers_1.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity divider_1 is
    port(DI : in  unsigned(7 downto 0);
         DO : out unsigned(7 downto 0));
end divider_1;

architecture archi of divider_1 is
begin

    DO <= DI / 2;

end archi;
```

定数 2 で除算する Verilog コード例

```
//  
// Division By Constant 2  
//  
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip  
// File: HDL_Coding_Techniques/dividers/dividers_1.v  
//  
module v_divider_1 (DI, DO);  
    input  [7:0] DI;  
    output [7:0] DO;  
  
    assign DO = DI / 2;  
  
endmodule
```

加算器、減算器、加減算器

XST では、次が認識されます。

- ・ 加算器

加算器は次の方法で記述できます。

- オプションのキャリー入力、および
- オプションのキャリー出力

- ・ 減算器

減算器は、オプションのボロー入力を付けて記述できます。

- ・ 加減算器

キャリー出力の記述

キャリー出力は通常、記述した加算の結果を最長のオペランドに追加されたビットを含む信号に代入して記述します。

キャリー出力を記述した VHDL コード例 1

```
input  [7:0]  A;
input  [7:0]  B;
wire   [8:0]  res;
wire                    carryout;

assign res = A + B;
assign carryout = res[8];
```

演算パッケージの確認

キャリー出力付き加算器を記述する場合、使用する演算パッケージに注意してください。選択した演算パッケージでは、加算器を記述する特定の使用方法が使用できないことがあります。

演算パッケージの確認例

- ・ 上記のコード例 1 は、std_logic_unsigned 演算パッケージでは動作しません。
- ・ これは、結果のサイズが最長の引数のサイズと同じになる必要があるためです。
- ・ このような場合、オペランドのサイズは次の例のように調整します。

キャリー出力を記述した VHDL コード例 2

```
signal A          : std_logic_unsigned(7 downto 0);
signal B          : std_logic_unsigned(7 downto 0);
signal res        : std_logic_unsigned(8 downto 0);
signal carryout   : std_logic;

res <= ("0" & A) + ("0" & B);
carryout <= res[8];
```

オペランドを整数型に変換

- ・ オペランドのサイズを調整するだけでなく、次も実行できます。
 1. オペランドを integer 型に変換します。
 2. その結果を std_logic_vector に変換し戻します。
- ・ 次のコード例 3 では、次が実行されています。
 - conv_std_logic_vector 変換関数は、std_logic_arith 演算パッケージに含まれています。
 - 符号なしの + 演算は、std_logic_unsigned 演算パッケージに含まれています。

キャリー出力を記述した VHDL コード例 3

```
signal A          : std_logic_vector(7 downto 0);
signal B          : std_logic_vector(7 downto 0);
signal res        : std_logic_vector(8 downto 0);
signal carryout   : std_logic;

res <= conv_std_logic_vector((conv_integer(A) + conv_integer(B)), 9);
carryout <= res[8];
```

加算器、減算器、加減算器のインプリメンテーション

スタンドアロンの加算器、減算器、加減算器には、次のような特徴があります。

- ・ DSP ブロックには自動的にインプリメントされません。
- ・ キャリー ロジックを使用して合成されます。

DSP48 ブロックのインプリメンテーション

- ・ 単純な加算器、減算器、加減算器を DSP ブロックにインプリメントするには、[DSP ブロックの使用 \(USE_DSP48\)](#) 制約の値を yes にします。
- ・ XST では、DSP48 ブロックに 1 レベルの出力レジスタがサポートされます。キャリーインまたは加減算のセレクトにレジスタが付いている場合も、これらのレジスタが DSP48 に挿入されます。
- ・ XST では、インプリメンテーションに必要な DSP リソースが 1 つのみの場合に、加減算器を DSP48 ブロックにインプリメントできます。1 つの DSP48 にフィットしない場合は、スライス ロジックを使用してマクロ全体がインプリメントされます。
- ・ DSP48 へのマクロのインプリメンテーションは、[DSP 使用率 \(DSP_UTILIZATION_RATIO\)](#) 制約をデフォルト値の auto に設定して制御します。
 - 加減算器はフィルターのようなより複雑なマクロの一部の場合、XST はこれを DSP ブロックに配置します。
 - 加減算器は複雑なマクロの一部ではない場合、LUT を使用してインプリメントされます。
- ・ これらのマクロを強制的に DSP48 に挿入するには、[DSP48 の使用 \(USE_DSP48\)](#) の値を yes に設定する必要があります。
 - DSP ブロックに加減算器を配置する際、XST ではそこからほかの DSP チェーンへの接続があるかどうかを確認されます。
 - 加減算器がほかの DSP チェーンへ接続される場合、XST では次が実行されます。
 - ◆ 高速の DSP 接続を利用しようとします。
 - ◆ これらの高速接続を使用してこの加減算器を DSP チェーンに接続します。
 - DSP48 ブロックに加減算器をインプリメントすると、XST では DSP48 リソースが自動的に制御されます。

最大マクロ コンフィギュレーション

- ・ 最適なパフォーマンスにするために、XST では次が実行されます。
 - マクロ コンフィギュレーションを最大限に推論、インプリメントしようとします。
 - DSP48 にできるだけ多くのレジスタを含めます。
- ・ マクロを特定のコンフィギュレーションにする場合は、[キープ](#) 制約を使用する必要があります。たとえば、DSP48 の 1 段目のレジスタを DSP48 に挿入しない場合は、KEEP 制約をこれらのレジスタの出力に設定する必要があります。

加算器、減算器、加減算器の関連制約

- ・ [DSP ブロックの使用](#)
- ・ [DSP 使用率](#)
- ・ [キープ](#)

加算器、減算器、加減算器のレポート

このセクションでは、加算器、減算器、加減算器のレポートについて説明します。

キャリー入力付き加算器

キャリー入力付き加算器には、次のような特徴があります。

- ・ 2 つの別々の加算器マクロが最初に推論されます。
- ・ HDL 合成 (HDL Synthesis 部分) でレポートされます。
- ・ アドバンス HDL 合成段階でキャリー入力付きの 1 つの加算器マクロにまとめられます。
- ・ マクロはアドバンス HDL 合成レポートに表示されます。

ボー入力付き減算器

ボー入力付き減算器には、次のような特徴があります。

- ・ 2 つの別々の減算器マクロが最初に推論されます。
- ・ マクロはアドバンス HDL 合成中にまとめられます。
- ・ キャリー出力ロジックについてはレポートされません。

加算器、減算器、加減算器のレポート例

```
=====
*                               HDL Synthesis                               *
=====

Synthesizing Unit <example>.
  Found 8-bit adder for signal <sum> created at line 9.
  Summary:
    inferred   1 Adder/Subtractor(s).
Unit <example> synthesized.

=====
HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                : 1
  8-bit adder                       : 1
=====
```

加算器、減算器、加減算器のコード例

アップデート情報は、「はじめに」のコード例を参照してください。

符号なし 8 ビット加算器の VHDL コード例

```
--
-- Unsigned 8-bit Adder
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/adders/adders_1.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity adders_1 is
    generic (
        WIDTH : integer := 8);
    port (
        A, B : in  std_logic_vector(WIDTH-1 downto 0);
        SUM  : out std_logic_vector(WIDTH-1 downto 0));
end adders_1;

architecture archi of adders_1 is
begin
    SUM <= A + B;
end archi;
```

キャリー イン付き符号なし 8 ビット加算器の Verilog コード例

```
//
// Unsigned 8-bit Adder with Carry In
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/adders/adders_2.v
//
module v_adders_2 (A, B, CI, SUM);

    parameter WIDTH = 8;
    input  [WIDTH-1:0] A;
    input  [WIDTH-1:0] B;
    input          CI;
    output [WIDTH-1:0] SUM;

    assign SUM = A + B + CI;

endmodule
```

乗算器

XST では、ソース コードの積演算子から乗算器マクロが推論されます。

- ・ 結果の出力信号のビット数は 2 つのオペランドの合計ビット数と等しくなります。たとえば、16 ビットの信号と 8 ビットの信号が乗算されると、結果は 24 ビットになります。
- ・ デバイスの全 MSB (最上位ビット) を使用しない場合、特に乗算器マクロをスライス ロジックにインプリメントする場合には、オペランドのビット数を必要最小限まで削減してみてください。

乗算器のインプリメンテーション

- ・ 乗算器は、次にインプリメントできます。
 - スライス ロジック
 - DSP ブロック
- ・ 乗算器をスライス ロジックにインプリメントするか DSP ブロック リソースにインプリメントするかは、[DSP ブロックの使用 \(USE_DSP48\)](#) 制約を auto に設定すると制御されます。
- ・ auto モードの場合、次が実行されます。
 - XST では、乗算器のオペランドが最小サイズである場合、DSP ブロック リソースに乗算器をインプリメントしようとします。最小サイズは、ターゲット デバイス ファミリーによって異なります。
 - XST では実際に使用可能な DSP ブロック リソースが考慮され、デバイスがオーバーマップされないようになります。XST では、使用可能な DSP ブロック リソースがすべて使用される可能性があります。[DSP 使用率 \(DSP_UTILIZATION_RATIO\)](#) を使用すると、これらのリソースを割り当てないままの状態にしておくことができます。
- ・ 乗算器をスライス ロジックまたは DSP ブロックに指定してインプリメントするには、該当する信号、エンティティ、モジュールで [DSP ブロックの使用 \(USE_DSP48\)](#) 制約を次のいずれかに設定します。
 - no (スライス ロジック)
 - yes (DSP ブロック)

DSP ブロックのインプリメンテーション

- ・ 乗算器を 1 つの DSP ブロックにインプリメントする場合、XST は DSP ブロックの次のようなパイプライン機能を利用しようとします。XST では、次の 2 レベルのレジスタまで吸収できます。
 - 乗算オペランド
 - 乗算の後ろ
- ・ 乗算器が 1 つの DSP ブロックに収まらない場合、XST はこのマクロを分解して、インプリメントします。この場合、次のいずれかにインプリメントされます。
 - 複数の DSP ブロック
 - DSP ブロックとスライス ロジックの両方
- ・ インプリメンテーションは、次のいずれかにします。
 - オペランドのサイズで駆動
 - 最大のパフォーマンスを目標
- ・ 複数の DSP ブロックへのインプリメンテーションは、パイプラインを実行することで改善されることがあります。この場合、**乗算器スタイル (MULT_STYLE)** を pipe_block に指定します。
- ・ XST は、パイプライン中にその乗算器のパフォーマンスを最大にするために必要な理想的なレジスタのステージ数を計算します。
 - 理想的な数のレジスタレベルが使用可能な場合、XST は目標を達成するためにそのレジスタを移動します。
 - 理想的な数のレジスタレベルが使用できない場合は、次のメッセージが表示されます。

```
INFO:Xst:2385 - HDL ADVISOR - You can improve the performance of the
multiplier Mmult_n0005 by adding 2 register level(s).
```

- ・ このメッセージで提案されるレジスタのステージ数を追加できます。
- ・ **キープ (KEEP)** を使用すると、レジスタが DSP ブロックに吸収されないようにできます。たとえば、乗算器のオペランドにあるレジスタが DSP ブロックに吸収されないようにするには、レジスタ出力にキープ制約を使用します。

スライス ロジックのインプリメンテーション

DSP ブロックの使用 (USE_DSP48) が auto に設定されると、次のような場合にほとんどの乗算器が DSP ブロック リソースにインプリメントされます。

- ・ 1 ステージまたは複数のレイテンシ ステージが使用可能、さらに
- ・ レイテンシ ステージが使用可能な DSP ブロックの制限内にある

スライス ロジックへの乗算器の強制インプリメンテーション

- ・ 乗算器をスライス ロジックに強制的にインプリメントするには、DSP ブロックの使用 (USE_DSP48) 制約の値を no に指定します。
- ・ 乗算器がスライス ロジックにインプリメントされる場合、XST では次が実行されます。
 - その演算子の周りでパイプラインができるかどうかを探します。
 - これらのレジスタを移動してデータ パス長を削減します。

パイプラインを使用した大型乗算器のパフォーマンス改善

パイプラインを使用すると大型乗算器のパフォーマンスをかなり改善できます。

- ・ パイプライン化の効果は、フリップフロップのリタイミングと同様です。
- ・ パイプライン ステージを挿入するには
 1. レジスタを記述します。
 2. それらのレジスタを乗算器の後に配置します。
 3. [乗算器スタイル \(MULT_STYLE\)](#) 制約を pipe_lut に設定します。

定数への乗算

- ・ XST では、乗算の引数の 1 つが定数の場合、次のどちらかの専用インプリメンテーション方法が使用されます。
 - CCM (Constant Coefficient Multiplier)
 - CSD (Canonical Signed Digit)
- ・ これらの方法は、乗算がスライス ロジックにインプリメントされた場合にのみ使用できます。
- ・ 最適化レベルは、その定数オペランドの特徴によって異なります。
 - CCM インプリメンテーションがデフォルトのスライス ロジックのインプリメンテーションよりも劣ることがあります。
 - XST では CCM か通常の乗算インプリメンテーションが選択されるようになっています。
- ・ CSD は自動的に選択されません。[乗算器スタイル](#)制約を使用して次を強制的に使用します。
 - CSD インプリメンテーション
 - CCM インプリメンテーション
- ・ XST では次の場合、CCM または CSD インプリメンテーションが使用されません。
 - 乗算が符号付きの場合
 - オペランドの 1 つが 32 ビットより大きい場合

乗算器の関連制約

- ・ [DSP ブロックの使用](#)
- ・ [DSP 使用率](#)
- ・ [キープ](#)
- ・ [乗算器スタイル](#)

乗算器の関連制約

- ・ 乗算器は HDL 合成中に推論されます。
- ・ 乗算器マクロによるレジスタの吸収は、アドバンス HDL 合成で発生します。次のレポート例を参照してください。

乗算器のレポート例

=====

```

*                               HDL Synthesis                               *
=====

```

```

Synthesizing Unit <v_multipliers_11>.
  Found 8-bit register for signal <rB>.
  Found 24-bit register for signal <RES>.
  Found 16-bit register for signal <rA>.
  Found 16x8-bit multiplier for signal <n0005> created at line 20.
  Summary:
inferred   1 Multiplier(s).
inferred  48 D-type flip-flop(s).
Unit <v_multipliers_11> synthesized.

```

```

=====
HDL Synthesis Report

```

```

Macro Statistics
# Multipliers                               : 1
  16x8-bit multiplier                       : 1
# Registers                                 : 3
  16-bit register                           : 1
  24-bit register                           : 1
  8-bit register                           : 1

```

```

=====
*                               Advanced HDL Synthesis                               *
=====

```

```

Synthesizing (advanced) Unit <v_multipliers_11>.
  Found pipelined multiplier on signal <n0005>:
    - 1 pipeline level(s) found in a register connected to the multiplier
macro output.
    Pushing register(s) into the multiplier macro.

    - 1 pipeline level(s) found in a register on signal <rA>.
    Pushing register(s) into the multiplier macro.

    - 1 pipeline level(s) found in a register on signal <rB>.
    Pushing register(s) into the multiplier macro.
INFO:Xst:2385 - HDL ADVISOR - You can improve the performance of the
multiplier Mmult_n0005 by adding 1 register level(s).
Unit <v_multipliers_11> synthesized (advanced).

```

```

=====
Advanced HDL Synthesis Report

```

```

Macro Statistics
# Multipliers                               : 1

```

16x8-bit registered multiplier : 1

=====

乗算器のコード例

アップデート情報は、「はじめに」のコード例を参照してください。

符号なし 8X4 ビット乗算器の VHDL コード例

```
--
-- Unsigned 8x4-bit Multiplier
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/multipliers/multipliers_1.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity multipliers_1 is
    generic (
        WIDTHA : integer := 8;
        WIDTHB : integer := 4);
    port (
        A      : in  std_logic_vector(WIDTHA-1 downto 0);
        B      : in  std_logic_vector(WIDTHB-1 downto 0);
        RES    : out std_logic_vector(WIDTHA+WIDTHB-1 downto 0));
end multipliers_1;

architecture beh of multipliers_1 is
begin
    RES <= A * B;
end beh;
```

符号なし 32x24 ビット乗算器の Verilog コード例

```
//
// Unsigned 32x24-bit Multiplier
//      1 latency stage on operands
//      3 latency stage after the multiplication
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/multipliers/multipliers_11.v
//
module v_multipliers_11 (clk, A, B, RES);

    parameter WIDTHA = 32;
    parameter WIDTHB = 24;
    input      clk;
    input  [WIDTHA-1:0]  A;
    input  [WIDTHB-1:0]  B;
    output [WIDTHA+WIDTHB-1:0] RES;

    reg  [WIDTHA-1:0]  rA;
    reg  [WIDTHB-1:0]  rB;
    reg  [WIDTHA+WIDTHB-1:0]  M  [3:0];
    integer i;

    always @(posedge clk)
    begin
        rA <= A;
        rB <= B;
        M[0] <= rA * rB;
        for (i = 0; i < 3; i = i+1)
            M[i+1] <= M[i];
        end
        assign RES = M[3];
    end

endmodule
```

乗加算と乗累算

- ・ 次のマクロはアドバンス HDL 合成中に推論されます。
 - 乗加算
 - 乗減算
 - 乗加減算
 - 乗累算
- ・ これらのマクロは、次が集められて推論されます。
 - 乗算器
 - 加減算器
 - レジスタはこれより前の HDL 合成中に推論されます。

乗加算と乗累算のインプリメンテーション

- ・ 乗加算と乗累算のインプリメンテーション中には、次が実行されます。
 - 推論された乗加算または乗累算マクロを DSP ブロック リソースにインプリメントできます。
 - DSP ブロックのパイプライン機能を利用しようとします。
 - 次にプルアップします。
 - ◆ 乗算オペランドの 2 レジスタ ステージ
 - ◆ 乗算の後ろの 1 レジスタ ステージ
 - ◆ 加算器、減算器、加算器/減算器の後ろの 1 レジスタ ステージ
 - ◆ 加算/減算の選択信号の 1 レジスタ ステージ
 - ◆ 加算器のオプションのキャリー入力の 1 レジスタ ステージ
 - XST では、インプリメンテーションに必要な DSP リソースが 1 つのみの場合に乗累算 (MAC) を DSP48 ブロックにインプリメントできます。
- ・ マクロが 1 つの DSP48 にフィットしない場合は、次のように処理されます。
 - 乗算器とアキュムレーター (累算) が別々のマクロとして処理されます。
 - 各マクロごとに個別に決定がされます。

DSP ブロック リソースへのマクロ インプリメンテーション

DSP ブロック リソースのマクロ インプリメンテーションは、[DSP ブロックの使用 \(USE_DSP48\)](#) 制約を auto に設定して制御します。

- ・ auto モードの場合、次が実行されます。
 - 乗加算および乗累算マクロをインプリメントします。
 - ターゲット デバイスで使用可能な DSP ブロック リソースが考慮されます。
 - 使用可能な DSP ブロック リソースがすべて使用される可能性があります。
- ・ [DSP 使用率 \(DSP_UTILIZATION_RATIO\)](#) を使用すると、これらのリソースを割り当てないままの状態にしておくことができます。
- DSP ブロックのすべてのパイプライン機能を利用することで、回路パフォーマンスを最大限にしようとします。
- レジスタを乗加算または乗累算マクロにできるだけ吸収させます。
- ・ [キープ \(KEEP\)](#) を使用すると、レジスタが DSP ブロックに吸収されないようにできます。たとえば、乗算器のオペランドにあるレジスタが DSP ブロックに吸収されないようにするには、レジスタ出力に `KEEP` を使用します。

乗加算と乗累算の関連制約

- ・ [DSP ブロックの使用](#)
- ・ [DSP 使用率](#)
- ・ [キープ](#)

乗加算と乗累算のレポート

XST からは、HDL Synthesis 段階で推論された乗算器、アキュムレーター、およびレジスタの詳細がレポートされます。

- ・ これらのマクロの乗加算または乗累算マクロへの合成情報については、Advanced HDL Synthesis セクションに記述されます。
- ・ どちらのマクロ タイプも、統合された MAC を表す部分の下に含まれます。

乗加算と乗累算のレポート例

```
=====
*                               HDL Synthesis                               *
=====

Synthesizing Unit <v_multipliers_7a>.
  Found 16-bit register for signal <accum>.
  Found 16-bit register for signal <mult>.
  Found 16-bit adder for signal <n0058> created at line 26.
  Found 8x8-bit multiplier for signal <n0005> created at line 18.
  Summary:
inferred   1 Multiplier(s).
inferred   1 Adder/Subtractor(s).
inferred  32 D-type flip-flop(s).
Unit <v_multipliers_7a> synthesized.
```

=====

HDL Synthesis Report

```

Macro Statistics
# Multipliers                      : 1
  8x8-bit multiplier              : 1
# Adders/Subtractors              : 1
  16-bit adder                   : 1
# Registers                       : 2
  16-bit register                : 2

```

=====

```

*                               Advanced HDL Synthesis                               *
=====

```

Synthesizing (advanced) Unit <v_multipliers_7a>.

The following registers are absorbed into accumulator <accum>: 1 register on signal <accum>.

Multiplier <Mmult_n0005> in block <v_multipliers_7a> and accumulator <accum> in block <v_multipliers_7a> are combined into a MAC<Mmac_n0005>.

The following registers are also absorbed by the MAC: <mult> in block <v_multipliers_7a>.

Unit <v_multipliers_7a> synthesized (advanced).

=====

Advanced HDL Synthesis Report

```

Macro Statistics
# MACs                             : 1
  8x8-to-16-bit MAC                : 1

```

=====

乗加算と乗累算のコード例

アップデート情報は、「はじめに」のコード例を参照してください。

乗算アップ アキュムレーター (乗算後にレジスタ付き) の VHDL コード例

```
--
-- Multiplier Up Accumulate with Register After Multiplication
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/multipliers/multipliers_7a.vhd
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity multipliers_7a is
    generic (p_width: integer:=8);
    port (clk, reset: in std_logic;
          A, B: in std_logic_vector(p_width-1 downto 0);
          RES: out std_logic_vector(p_width*2-1 downto 0));
end multipliers_7a;

architecture beh of multipliers_7a is
    signal mult, accum: std_logic_vector(p_width*2-1 downto 0);
begin

    process (clk)
    begin
        if (clk'event and clk='1') then
            if (reset = '1') then
                accum <= (others => '0');
                mult <= (others => '0');
            else
                accum <= accum + mult;
                mult <= A * B;
            end if;
        end if;
    end process;

    RES <= accum;

end beh;
```

乗算器アップ アキュムレーターの Verilog コード例

```
//
// Multiplier Up Accumulate with:
//   Registered operands
//   Registered multiplication
//   Accumulation
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/multipliers/multiply_accum_2.v
//
module v_multiply_accum_2 (clk, rst, A, B, RES);

    parameter WIDTH = 8;
    input          clk;
    input          rst;
    input  [WIDTH-1:0]  A, B;
    output [2*WIDTH-1:0] RES;

    reg  [WIDTH-1:0]  rA, rB;
    reg  [2*WIDTH-1:0] mult, accum;

    always @(posedge clk)
    begin
        if (rst) begin
            rA    <= {WIDTH{1'b0}};
            rB    <= {WIDTH{1'b0}};
            mult   <= {2*WIDTH{1'b0}};
            accum  <= {2*WIDTH{1'b0}};
        end
    else begin
        rA <= A;
        rB <= B;
        mult <= rA * rB;
        accum <= accum + mult;
    end
    end
    assign RES = accum;

endmodule
```

DSP の推論

XST には、ポータブルのビヘイビアー ソース コードを含むフィルターを記述するため、さらに拡張された推論機能が含まれます。

- ・ 次の基本的なファンクションのより精密な推論機能も含まれます。
 - レイテンシ ステージ (レジスタ)
 - 乗算
 - 乗加算/減算
 - 累算
 - 乗累積
 - ROM
- ・ XST では、次が実行され、高パフォーマンスのインプリメンテーションおよび電力削減が達成できるようになっています。
 - 基本的な論理エレメント間の文脈関係があるかどうかを判断
 - ザイリンクス デバイスで使用可能な次の DSP ブロック リソースの優れた機能を利用
 - ◆ パイプライン ステージ
 - ◆ カスケード パス
 - ◆ 前置加算器ステージ
 - ◆ 時分割多重化
- ・ DSP ブロックの機能を適切に使用するには、加算器ツリーではなく、加算器チェーンをフィルター記述の骨組みとして使用します。VHDL の for-generate 文のような HDL 言語の中には、この方法でフィルターを記述させて、コードの可読性と拡張性を最大限にするものもあります。
- ・ DSP ブロック リソースの詳細は、次を参照してください。
 - [『Virtex-6 FPGA DSP48E1 スライス ユーザー ガイド』\(UG369\) \(ザイリンクス サポート ウェブサイトより\)](#)
 - [『Spartan-6FPGA DSP48A1 スライス ユーザー ガイド』\(UG389\) \(ザイリンクス サポート ウェブサイトより\)](#)

対称フィルター

ザイリンクス DSP ブロックのオプションの前置加算器は、対称フィルター用に設計されています。対称係数フィルターを記述する場合は、前置加算器を使用して必要な DSP ブロックの数を半分に減らします。

- ・ フィルターを一般的な方法で記述しないでください。
 - XST では自動的に対称係数が識別されて因数分解されません。
 - 前置加算器を使用し、DSP ブロックが適宜にコンフィギュレーションされるようにするには、その因数分解形式を手動でコード記述する必要があります。
 - 次の SymSystolicFilter および SymTransposeConvFilter コード例は、その具体例です。
- ・ 前置加算器の機能を使用するには、XST でその前置加算器の容量と同じ記述が（データ幅がそれより小さくても）識別される必要があります。
 - この要件は、前置加算器にのみ該当します。
 - 前置加算オペランドの明示的なパディングに符合付きまたは符号なしの拡張を使用して、DSP リソースの推論およびインプリメンテーションが正しく実行されるようにしてください。次の例を参照してください。

明示的なデータ拡張を使用した前置加算器のコード例

```
--
-- Explicit padding of pre-adder operands
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/dsp/preadder_padding.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity preadder_padding is

    generic (
        DATA_WIDTH : integer := 16
    );
    port (
        clk : in std_logic;
        a : in signed(DATA_WIDTH-1 downto 0);
        b : in signed(DATA_WIDTH-1 downto 0);
        c : in signed(DATA_WIDTH-1 downto 0);
        d : in signed(2*DATA_WIDTH-1 downto 0);
        o : out signed(2*DATA_WIDTH-1 downto 0)
    );

end preadder_padding;

architecture behavioral of preadder_padding is

    constant PREADD_WIDTH : integer := 18;
```

```
signal a_resized : signed(PREADD_WIDTH-1 downto 0);
signal b_resized : signed(PREADD_WIDTH-1 downto 0);
signal pre       : signed(PREADD_WIDTH-1 downto 0);
signal m         : signed(DATA_WIDTH+PREADD_WIDTH-1 downto 0);
signal p         : signed(DATA_WIDTH+PREADD_WIDTH-1 downto 0);

begin

    assert DATA_WIDTH <= PREADD_WIDTH report "DATA_WIDTH exceeds limit of 18 bits" severity ERROR;

    a_resized <= RESIZE(a, PREADD_WIDTH);
    b_resized <= RESIZE(b, PREADD_WIDTH);

    process (clk)
    begin
        if rising_edge(clk) then
            pre <= a_resized + b_resized;
            m <= pre * c;
            p <= m + d;
        end if;
    end process;

    o <= RESIZE(p, 2*DATA_WIDTH);

end behavioral;
```

DSP の推論のコード例

アップデート情報は、「はじめに」のコード例を参照してください。

DSP リファレンス デザイン

デザイン	HDL 言語 の指定	説明	デバイス
PolyDecFilter	VHDL	多層間引きフィルター	Spartan®-6 Virtex®-6 7 シリーズ
PolyIntrpFilter	VHDL	多層補間フィルター	Spartan-6 Virtex-6 7 シリーズ
EvenSymSystFIR	VHDL	偶数タップ付き対称シストリック フィルター。DSP ブロックの前置加算器の利点を生かすために、対称係数が因数分解されています。	Virtex-6 7 シリーズ
OddSymSystFIR	VHDL	奇数タップ付き対称シストリック フィルター。DSP ブロックの前置加算器の利点を生かすために、対称係数が因数分解されています。	Virtex-6 7 シリーズ
EvenSymTranspConvFIR	VHDL	偶数タップ付き対称転置型たたみ込みフィルター。DSP ブロックの前置加算器の利点を生かすために、対称係数が因数分解されています。	Virtex-6 7 シリーズ
OddSymTranspConvFIR	VHDL	奇数タップ付き対称転置型たたみ込みフィルター。DSP ブロックの前置加算器の利点を生かすために、対称係数が因数分解されています。	Virtex-6 7 シリーズ
AlphaBlender	VHDL Verilog	前置加算器、乗算器、後置加算器の利点を生かすため、画像合成でよく使用される α (アルファ) ブレンディング関数を 1 つの DSP ブロックにインプリメントします。	Spartan-6 Virtex-6 7 シリーズ
ComplexMult	VHDL	複雑な乗算器を記述する単純な方法です。	Spartan-6 Virtex-6 7 シリーズ
ComplexMultAcc	VHDL	複雑な乗累算機能を記述する単純な方法です。	Spartan-6 Virtex-6 7 シリーズ

リソース共有

XST には、「リソース共有」と呼ばれる高レベルの最適化機能が含まれています。

- ・ リソースを共有すると、演算子の数を最小限に抑えることができるので、デバイス使用率を削減できます。
- ・ リソース共有は、2 つの類似する四則演算子が、それらに対応する出力が同時に使用されることがない場合、デバイスの共有リソースを使用してインプリメントできるという原則に基づいて実行されます。
- ・ リソース共有では、通常因数分解された入力間を選択するために、マルチプレクサ ロジックが追加で作成されます。因数分解は、このロジックを最小にする方法で実行されます。
- ・ リソース共有は、どの最適化手法を選択したとしても、デフォルトでイネーブルになります。

XST のリソース共有サポート

XST では、次のリソースの共有がサポートされます。

- ・ 加算器
- ・ 減算器
- ・ 加減算器
- ・ 乗算器

リソース共有のディスエーブル

- ・ 次の場合は、リソース共有をディスエーブルにすることをお勧めします。
 - 回路パフォーマンスを主な最適化目標としてする場合 および
 - タイミング目標を達成できない場合
- ・ リソース共有が実行されると、HDL Advisor のメッセージでそれが表示されます。

リソース共有の関連制約

リソース共有

リソース共有のレポート

演算のリソース共有には、次のような特徴があります。

- ・ HDL 合成中に実行されます。
- ・ 次に表示されます。
 - 演算マクロの統計
 - HDL Advisor メッセージ

リソース共有のレポート例

```

=====
*                               HDL Synthesis                               *
=====

Synthesizing Unit <resource_sharing_1>.
  Found 8-bit adder for signal <n0017> created at line 18.
  Found 8-bit subtractor for signal <n0004> created at line 18.
  Found 8-bit 2-to-1 multiplexer for signal <RES> created at line 18.
  Summary:
    inferred   1 Adder/Subtractor(s).
    inferred   1 Multiplexer(s).
Unit <resource_sharing_1> synthesized.

=====
HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                : 1
  8-bit addsub                      : 1
# Multiplexers                      : 1
  8-bit 2-to-1 multiplexer          : 1

=====
INFO:Xst:1767 - HDL ADVISOR - Resource sharing has identified that some
arithmetic operations in this design can share the same physical
resources for reduced device utilization.
For improved clock frequency you may try to disable resource sharing.

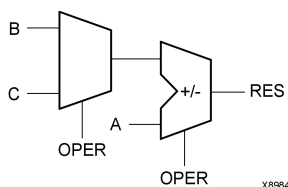
```

リソース共有のコード例

アップデート情報は、「はじめに」のコード例を参照してください。

次のコード例は、次の図のような結果になります。

リソース共有の図



リソース共有の VHDL コード例

```
--
-- Resource Sharing
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/resource_sharing/resource_sharing_1.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity resource_sharing_1 is
    port(A, B, C : in  std_logic_vector(7 downto 0);
         OPER    : in  std_logic;
         RES     : out std_logic_vector(7 downto 0));
end resource_sharing_1;

architecture archi of resource_sharing_1 is
begin

    RES <= A + B when OPER='0' else A - C;

end archi;
```

リソース共有の Verilog コード例

```
//
// Resource Sharing
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/resource_sharing/resource_sharing_1.v
//
module v_resource_sharing_1 (A, B, C, OPER, RES);
    input  [7:0] A, B, C;
    input  OPER;
    output [7:0] RES;
    wire   [7:0] RES;

    assign RES = !OPER ? A + B : A - C;

endmodule
```

RAM の HDL コード記述方法

XST では、ランダム アクセス メモリ (RAM) の推論により次を実現します。

- ・ 手動でザイリンクス RAM プリミティブをインスタンスエートする必要なし
- ・ 時間の節約
- ・ HDL ソース コードを移動および拡張しやすい

分散 RAM および専用ブロック RAM

- ・ RAM リソースには、次の 2 タイプがあります。
 - 分散 RAM
非同期読み出しが含まれる RAM 記述で使用する必要があります。
 - 専用ブロック RAM
同期読み出しが含まれる RAM 記述で通常使用します。
- ・ [RAM スタイル \(RAM_STYLE\)](#) は、RAM のインプリメンテーションを制御するために使用します。
- ・ 詳細については、次から分散 RAM と関連トピックを参照してください。
 - [『Virtex-6 FPGA メモリ リソース ユーザー ガイド』](#)
 - [『Virtex-6 FPGA コンフィギャブル ロジック ブロック ユーザー ガイド』](#)
 - [『Spartan-6 FPGA ブロック RAM リソース ユーザー ガイド』](#)
 - [『Spartan-6 FPGA コンフィギャブル ロジック ブロック ユーザー ガイド』](#)

分散 RAM と専用ブロック RAM の比較

どちらのリソースタイプでも、データは同時に RAM に記述されます。分散 RAM と専用ブロック RAM の主な違いは、RAM からのデータの読み出され方にあります。次の表に、各項目の詳細を示します。

動作	分散 RAM	専用ブロック RAM
書き込み	同期	同期
読み出し	非同期	同期

分散 RAM と専用ブロック RAM 間の選択

分散 RAM と専用ブロック RAM のどちらを使用するかは、次によって決定できます。

- ・ HDL で記述した RAM の特性
- ・ 特定のインプリメンテーション スタイルを指定したかどうか
- ・ 使用可能なブロック RAM リソース数

非同期読み出し (分散 RAM)

- ・ 非同期読み出しの RAM 記述には、次のような特徴があります。
 - 分散 RAM を使用してインプリメントされます。
 - 専用ブロック RAM にはインプリメントできません。
- ・ 分散 RAM は、適切にコンフィギュレーションされたスライス ロジックにインプリメントされます。

同期読み出し (専用ブロック RAM)

同期読み出しの RAM 記述には、次のような特徴があります。

- ・ 通常は専用ブロック RAM に含まれます。
- ・ 特別に指定した場合、またはデバイスリソース使用率を考慮した場合などは、分散 RAM にレジスタを追加してインプリメントもできます。

RAM でサポートされる機能

RAM でサポートされる機能は、次のとおりです。

- ・ [RAM の推論機能](#)
- ・ [パリティビット](#)

RAM の推論機能

RAM 推論機能には、次が含まれます。

- ・ どのサイズおよびデータ幅でもサポート。XST では、RAM 記述が 1 つまたは複数の RAM プリミティブへのマップされます。
 - ・ シングル ポート、単純なデュアル ポート、真のデュアル ポート
 - ・ 最大 2 つの書き込みポート
 - ・ 複数の読み出しポート
- 書き込みポートが 1 つしか記述されていない場合、XST ではその書き込みアドレスとは異なるアドレスで RAM 内容にアクセスする読み出しポートが複数付いた RAM 記述が認識できます。
- ・ 非対称ポートの付いたシンプルなデュアル ポートと真のデュアル ポート。詳細は、「[非対称ポートのサポート \(ブロック RAM\)](#)」を参照してください。
 - ・ 書き込みイネーブル
 - ・ RAM イネーブル (ブロック RAM)
 - ・ データ出力リセット (ブロック RAM)
 - ・ オプションの出力レジスタ (ブロック RAM)
 - ・ バイト幅書き込みイネーブル (ブロック RAM)
 - ・ 各 RAM ポートは、そのクロック、RAM イネーブル、書き込みイネーブル、データ出力リセットで制御できます。
 - ・ 初期内容の指定

パリティ ビット

XST では、パリティ ビットはサポートされません。

- ・ パリティ ビットは一部のブロック RAM プリミティブで使用可能です。
- ・ XST では、パリティ ビットを通常の変数ビットとして使用して、記述したデータ幅に対応できます。
- ・ XST では、次が実行できません。
 - パリティ制御ロジックを自動的に生成
 - これらのパリティ ビットを意図した目的に使用

RAM の HDL コード記述のガイドライン

RAM の HDL コード記述のガイドラインには、次が含まれます。

- ・ [RAM の記述](#)
- ・ [読み出しアクセスの記述](#)
- ・ [ブロック RAM の読み出し/書き込みの同期](#)
- ・ [リセット可能なデータ出力 \(ブロック RAM\)](#)
- ・ [バイト幅の書き込みイネーブルのサポート \(ブロック RAM\)](#)
- ・ [非対称ポートのサポート](#)
- ・ [RAM の初期内容](#)

RAM の記述

RAM は、通常配列オブジェクトの 1 配列を使用して記述されます。

VHDL での RAM の記述 (書き込みポート 1 つ)

1 つの書き込みポートが付いた RAM を記述するには、VHDL の「signal」を次のように使用します。

```
type ram_type is array (0 to 255) of std_logic_vector (15 downto 0);  
signal RAM : ram_type;
```

VHDL での RAM の記述 (書き込みポート 2 つ)

VHDL で 2 つの書き込みポートが付いた RAM を記述するには、「signal」ではなく「shared variable」を使用します。

```
type ram_type is array (0 to 255) of std_logic_vector (15 downto 0);  
shared variable RAM : ram_type;
```

- ・ 2 つの書き込みポートが付いた RAM を記述するのに「signal」を使用しようとする、XST で拒否されます。このような記述は、シミュレーションで正しく動作しません。
- ・ 「shared variable」は変数の延長で、プロセス間通信を可能にします。
 - 「shared variable」を使用する場合は変数よりも注意が必要です。
 - 「shared variable」はすべての基本特性を引き継ぎます。
 - 順次プロセスで記述されている順序で、記述された機能が調整されます。
 - 同じシミュレーション サイクルで共通変数 (shared variable) へ 2 つ以上のプロセスを代入すると、予測不可能な結果になることがあります。
- ・ RAM に 1 つしか書き込みポートがない場合、XST では「shared variable」を使用できませんが、お勧めしません。「signal」を使用してください。

Verilog での RAM の記述コード例

```
reg [15:0] RAM [0:255];
```

書き込みアクセスの記述

書き込みアクセスの記述には、次が含まれます。

- ・ [VHDL での書き込みアクセスの記述](#)
- ・ [Verilog での書き込みアクセスの記述](#)

VHDL での書き込みアクセスの記述

- ・ VHDL の「signal」で記述された RAM の場合、RAM への書き込みは次のように記述します。

```
process (clk)
begin
    if rising_edge(clk) then
        if we = '1' then
            RAM(conv_integer(addr)) <= di;
        end if;
    end if;
end process;
```

- ・ このアドレス信号は、通常次のように宣言されます。

```
signal addr : std_logic_vector(ADDR_WIDTH-1 downto 0);
```

std_logic_unsigned の含有

- ・ conv_integer 変換関数を使用するためには、std_logic_unsigned を含める必要があります。
- ・ std_logic_signed には conv_integer 関数も含まれていますが、このインスタンス内で使用するのをお勧めしません。
- ・ use std_logic_signed を使用すると、次が実行されます。
 - － XST では、アドレス信号は符号付きで記述されると仮定されます。
 - － すべての負の値が無視されます。
 - － 推論された RAM が必要なサイズの半分になることがあります。
- ・ 符号付きデータ記述が必要な部分がデザインにある場合は、RAM とは別の部分で記述してください。

VHDL の共通変数を使用した RAM の記述コード例

このコード例は、次の場合の通常書き込みを示しています。

- ・ RAM に書き込みポートが 2 つ付いていて、さらに
- ・ VHDL の「shared variable」を使用して記述されている場合

```
process (clk)
begin
    if rising_edge(clk) then
        if we = '1' then
            RAM(conv_integer(addr)) := di;
        end if;
    end if;
end process;
```


Verilog での書き込みアクセスの記述

```
always @ (posedge clk)
begin
  if (we)
    RAM[addr] <= di;
end
```

読み出しアクセスの記述

読み出しアクセスの記述には、次が含まれます。

- ・ [VHDL での読み出しアクセスの記述](#)
- ・ [Verilog での読み出しアクセスの記述](#)

VHDL での読み出しアクセスの記述

- ・ RAM は、通常指定したアドレス位置から次のように読み出されます。

```
do <= RAM( conv_integer(addr));
```
- ・ この文が単純な同時処理文であるか、シーケンシャル プロセスで記述されるかによって、次が決まります。
 - 読み出しが非同期か同期か
 - RAM コンポーネントは、次のいずれかを使用してインプリメントされます。
 - ◆ ブロック RAM リソース
 - ◆ 分散 RAM リソース
- ・ 詳細は、「[ブロック RAM の読み出し/書き込みの同期](#)」を参照してください。

ブロック RAM リソースへの RAM のインプリメントコード例

```
process (clk)
begin
  do <= RAM( conv_integer(addr));
end process;
```

Verilog での読み出しアクセスの記述

- ・ 非同期の読み出しは、assign 文で記述します。

```
assign do = RAM[addr];
```
- ・ 同期読み出しは、シーケンシャルな always ブロックで記述されます。

```
always @ (posedge clk)
begin
  do <= RAM[addr];
end
```
- ・ 詳細は、「[ブロック RAM の読み出し/書き込みの同期](#)」を参照してください。

ブロック RAM の読み出し/書き込みの同期

- ・ ブロック RAM リソースは、次の同期モードを提供するようにコンフィギュレーションできます。
 - read-first
新しい内容が読み込まれる前に、古い内容が読み出されます。
 - write-first
 - ◆ 新しい内容が即座に読み出せるようになります。
 - ◆ write-first (または read-through)
 - no-change
新しい内容が RAM に読み込まれる間、データ出力に変化はありません。
- ・ XST では、これらすべての同期モードの推論がサポートされます。RAM のポートごとに、異なる同期モードを記述できます。

ブロック RAM の読み出し/書き込みの同期の VHDL コード例 1

```
process (clk)
begin
    if (clk'event and clk = '1') then
        if (we = '1') then
            RAM(conv_integer(addr)) <= di;
        end if;
        do <= RAM(conv_integer(addr));
    end if;
end process;
```

ブロック RAM の読み出し/書き込みの同期の VHDL コード例 2

次のコード例では、write-first 同期ポートを記述しています。

```
process (clk)
begin
    if (clk'event and clk = '1') then
        if (we = '1') then
            RAM(conv_integer(addr)) <= di;
            do <= di;
        else
            do <= RAM(conv_integer(addr));
        end if;
    end if;
end process;
```

ブロック RAM の読み出し/書き込みの同期の VHDL コード例 3

次のコード例では、no-change 同期ポートを記述しています。

```
process (clk)
begin
    if (clk'event and clk = '1') then
        if (we = '1') then
            RAM(conv_integer(addr)) <= di;
        else
            do <= RAM(conv_integer(addr));
        end if;
    end if;
end process;
```

ブロック RAM の読み出し/書き込みの同期の VHDL コード例 4

注意： デュアルライト RAM を VHDL の共通変数を使用して記述する場合は、次の例のように同期が read-first ではなく、write-first で記述されます。

```
process (clk)
begin
    if (clk'event and clk = '1') then
        if (we = '1') then
            RAM(conv_integer(addr)) := di;
        end if;
        do <= RAM(conv_integer(addr));
    end if;
end process;
```

ブロック RAM の読み出し/書き込みの同期の VHDL コード例 5

read-first 同期を記述する場合は、プロセス文本体の順序を変更します。

```
process (clk)
begin
    if (clk'event and clk = '1') then
        do <= RAM(conv_integer(addr));
        if (we = '1') then
            RAM(conv_integer(addr)) := di;
        end if;
    end if;
end process;
```

リセット可能なデータ出力 (ブロック RAM)

オプションで、同期読み出しデータのどの定数値に対してもリセットを記述できます。

- ・ XST はリセットを認識し、ブロック RAM の同期セット/リセット機能を使用します。
- ・ read-first 同期を使用した RAM ポートの場合、リセット機能は次のコード例のように記述します。

リセット可能なデータ出力 (ブロック RAM) のコード例

```
process (clk)
begin
    if clk'event and clk = '1' then
        if en = '1' then -- optional RAM enable
            if we = '1' then -- write enable
                ram(conv_integer(addr)) <= di;
            end if;
            if rst = '1' then -- optional dataout reset
                do <= "00011101";
            else
                do <= ram(conv_integer(addr));
            end if;
        end if;
    end if;
end process;
```

バイト幅書き込みイネーブル (ブロック RAM)

ザイリンクスでは、ブロック RAM でのバイト幅の書き込みイネーブルをサポートしています。

- ・ ブロック RAM でのバイト幅の書き込みイネーブルを使用して、次を実行します。
 - RAM へのデータ書き込みを高度に制御
 - アドレス指定されたメモリーの書き込み可能な部分 8 ビットを別々に指定
- ・ HDL 記述と推論の点から考えると、この概念は列ベースの書き込みとして記述可能
 - RAM は、同じサイズの列の集まりとして表記されます。
 - 書き込みサイクル中は、これらの列ごとへの書き込みを別々に制御します。
- ・ XST の推論により、ブロック RAM のバイト幅の書き込みイネーブル機能の利点を生かすことができます。
- ・ XST では、次の 2 つの記述スタイルがサポートされます。
 - 1 プロセス記述スタイル (推奨)
 - 2 プロセス記述スタイル (推奨されない)

1 プロセス記述スタイル (推奨)

1 プロセス記述スタイルの方が 2 プロセス記述スタイルよりもエラーの要因が少なくなっています。

次の条件が満たされる場合、記述され RAM はバイト ライト イネーブル機能を使用して、ブロック RAM リソースにインプリメントされます。

- ・ 書き込み列の幅が同じ
- ・ 使用可能な書き込み列幅 : 8 ビット、9 ビット、16 ビット、18 ビット
5 ビットまたは 12 ビットなどのほかの列幅の場合、XST では分散 RAM リソースを使用して、データ入力にマルチプレクサ ロジックを追加します。
- ・ 書き込み列数 : 任意
- ・ RAM の深さ : 任意
XST は 1 つまたは複数のブロック RAM プリミティブを必要に応じ使用して RAM をインプリメントします。
- ・ サポートされる読み出し/書き込み同期 : read-first、write-first、no-change

1 プロセス記述スタイルの VHDL コード例

次のコード例では、ジェネリックと for-loop 文を使用し、必要な書き込み列数と列幅を簡単に変更可能なコンフィギュレーションになっています。

```
--
-- Single-Port BRAM with Byte-wide Write Enable
-- 2x8-bit write
-- Read-First mode
-- Single-process description
-- Compact description of the write with a for-loop statement
-- Column width and number of columns easily configurable
--
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/bytewrite_ram_1b.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity bytewrite_ram_1b is

    generic (
        SIZE      : integer := 1024;
        ADDR_WIDTH : integer := 10;
        COL_WIDTH  : integer := 8;
        NB_COL     : integer := 2);

    port (
        clk : in  std_logic;
        we  : in  std_logic_vector(NB_COL-1 downto 0);
        addr : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
        di   : in  std_logic_vector(NB_COL*COL_WIDTH-1 downto 0);
        do   : out std_logic_vector(NB_COL*COL_WIDTH-1 downto 0));

end bytewrite_ram_1b;
```

```
architecture behavioral of bytewrite_ram_1b is

    type ram_type is array (SIZE-1 downto 0)
        of std_logic_vector (NB_COL*COL_WIDTH-1 downto 0);
    signal RAM : ram_type := (others => (others => '0'));

begin

    process (clk)
    begin
        if rising_edge(clk) then
            do <= RAM(conv_integer(addr));
            for i in 0 to NB_COL-1 loop
                if we(i) = '1' then
                    RAM(conv_integer(addr)) ((i+1)*COL_WIDTH-1 downto i*COL_WIDTH)
                    <= di((i+1)*COL_WIDTH-1 downto i*COL_WIDTH);
                end if;
            end loop;
        end if;
    end process;

end behavioral;
```

1 プロセス記述スタイルの Verilog コード例

次のコード例では、パラメーターと generate-for 文を使用しています。

```
//
// Single-Port BRAM with Byte-wide Write Enable
// 4x9-bit write
// Read-First mode
// Single-process description
// Compact description of the write with a generate-for statement
// Column width and number of columns easily configurable
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/rams/bytewrite_ram_1b.v
//
module v_bytewrite_ram_1b (clk, we, addr, di, do);

    parameter SIZE = 1024;
    parameter ADDR_WIDTH = 10;
    parameter COL_WIDTH = 9;
    parameter NB_COL = 4;

    input      clk;
    input      [NB_COL-1:0] we;
    input      [ADDR_WIDTH-1:0] addr;
    input      [NB_COL*COL_WIDTH-1:0] di;
    output reg [NB_COL*COL_WIDTH-1:0] do;

    reg        [NB_COL*COL_WIDTH-1:0] RAM [SIZE-1:0];

    always @(posedge clk)
    begin
        do <= RAM[addr];
    end

    generate
    genvar i;
    for (i = 0; i < NB_COL; i = i+1)
    begin
        always @(posedge clk)
        begin
            if (we[i])
                RAM[addr][(i+1)*COL_WIDTH-1:i*COL_WIDTH] <= di[(i+1)*COL_WIDTH-1:i*COL_WIDTH];
            end
        end
    endgenerate
endmodule
```

1 プロセス記述スタイルの VHDL コード例 (NO_CHANGE モード)

1 プロセス記述スタイルは、NO_CHANGE 読み出し/書き込み同期モードを使用してバイト幅の書き込みイネーブル機能を適切に記述できる唯一の方法です。

```
--
-- Single-Port BRAM with Byte-wide Write Enable
-- 2x8-bit write
-- No-Change mode
-- Single-process description
-- Compact description of the write with a for-loop statement
-- Column width and number of columns easily configurable
--
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/bytewrite_nochange.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity bytewrite_nochange is

    generic (
        SIZE      : integer := 1024;
        ADDR_WIDTH : integer := 10;
        COL_WIDTH  : integer := 8;
        NB_COL     : integer := 2);

    port (
        clk : in  std_logic;
        we  : in  std_logic_vector(NB_COL-1 downto 0);
        addr : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
        di   : in  std_logic_vector(NB_COL*COL_WIDTH-1 downto 0);
        do   : out std_logic_vector(NB_COL*COL_WIDTH-1 downto 0));

end bytewrite_nochange;

architecture behavioral of bytewrite_nochange is

    type ram_type is array (SIZE-1 downto 0) of std_logic_vector (NB_COL*COL_WIDTH-1 downto 0);
    signal RAM : ram_type := (others => (others => '0'));
begin

    process (clk)
    begin
        if rising_edge(clk) then
            if (we = (we'range => '0')) then
                do <= RAM(conv_integer(addr));
            end if;
            for i in 0 to NB_COL-1 loop
                if we(i) = '1' then
                    RAM(conv_integer(addr)) ((i+1)*COL_WIDTH-1 downto i*COL_WIDTH)
                    <= di((i+1)*COL_WIDTH-1 downto i*COL_WIDTH);
                end if;
            end loop;
        end if;
    end process;

end behavioral;
```


1 プロセス記述スタイルの Verilog コード例 (NO_CHANGE モード)

```
//  
// Single-Port BRAM with Byte-wide Write Enable  
// 4x9-bit write  
// No-Change mode  
// Single-process description  
// Compact description of the write with a generate-for statement  
// Column width and number of columns easily configurable  
//  
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip  
// File: HDL_Coding_Techniques/rams/bytewrite_nochange.v  
//  
module v_bytewrite_nochange (clk, we, addr, di, do);  
  
    parameter SIZE = 1024;  
    parameter ADDR_WIDTH = 10;  
    parameter COL_WIDTH = 9;  
    parameter NB_COL = 4;  
  
    input      clk;  
    input      [NB_COL-1:0] we;  
    input      [ADDR_WIDTH-1:0] addr;  
    input      [NB_COL*COL_WIDTH-1:0] di;  
    output reg [NB_COL*COL_WIDTH-1:0] do;  
  
    reg        [NB_COL*COL_WIDTH-1:0] RAM [SIZE-1:0];  
  
    always @(posedge clk)  
    begin  
        if (~|we)  
            do <= RAM[addr];  
    end  
  
    generate  
    genvar i;  
    for (i = 0; i < NB_COL; i = i+1)  
    begin  
        always @(posedge clk)  
        begin  
            if (we[i])  
                RAM[addr][(i+1)*COL_WIDTH-1:i*COL_WIDTH]  
                <= di[(i+1)*COL_WIDTH-1:i*COL_WIDTH];  
        end  
    end  
endgenerate  
  
endmodule
```

2 プロセス記述スタイル

ブロック RAM のバイト幅書き込みイネーブル機能の利点を生かすためには、適切なデータ読み出し同期を記述しないと、XST が分散 RAM リソースを使用して記述した機能をインプリメントしてしまいます。

- ・ この記述スタイルはサポートはされますが、お勧めできる方法ではありません。
- ・ 2 プロセス記述スタイルでは、NO_CHANGE 同期モードを使用してバイト幅の書き込みイネーブル機能を適切に記述できません。
- ・ ザイリンクスでは、次を推奨しています。
 - 現在 2 プロセス記述スタイルを使用している場合は、1 プロセス記述スタイルへ変更してみてください。
 - 新しいデザインには、2 プロセス記述スタイルは使用しないでください。
- ・ コードを 1 プロセス記述スタイルに変更できない場合でも、XST では 2 プロセスの記述スタイルがサポートされています。
- ・ 2 プロセス記述スタイル
 - どのデータが読み込まれて、読み出されるのかバイトごとに記述される組み合わせプロセス。特に書き込みイネーブル機能は、メインの順次プロセスではなく、このプロセスで記述します。
 - 書き込みおよび読み出し同期を記述する順次プロセス
 - データ幅は 1 プロセスの方法よりも限られる
 - ◆ 書き込み列数 : 2 または 4
 - ◆ 書き込み列幅 : 8 ビットまたは 9 ビット
 - ◆ サポートされるデータ幅 : 2x8 ビット (各 8 ビットが 2 列)、2x9 ビット、4x8 ビット、4x9 ビット

2 プロセス記述スタイルの VHDL コード例

```
--
-- Single-Port BRAM with Byte-wide Write Enable
-- 2x8-bit write
-- Read-First Mode
-- Two-process description
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/rams_24.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_24 is

    generic (
        SIZE      : integer := 512;
        ADDR_WIDTH : integer := 9;
        COL_WIDTH  : integer := 16;
        NB_COL     : integer := 2);

    port (
        clk : in  std_logic;
        we  : in  std_logic_vector(NB_COL-1 downto 0);
        addr : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
        di   : in  std_logic_vector(NB_COL*COL_WIDTH-1 downto 0);
        do   : out std_logic_vector(NB_COL*COL_WIDTH-1 downto 0));

end rams_24;

architecture syn of rams_24 is

    type ram_type is array (SIZE-1 downto 0) of std_logic_vector (NB_COL*COL_WIDTH-1 downto 0);
    signal RAM : ram_type := (others => (others => '0'));

    signal di0, di1 : std_logic_vector (COL_WIDTH-1 downto 0);
begin

    process(we, di)
    begin
        if we(1) = '1' then
            di1 <= di(2*COL_WIDTH-1 downto 1*COL_WIDTH);
        else
            di1 <= RAM(conv_integer(addr))(2*COL_WIDTH-1 downto 1*COL_WIDTH);
        end if;

        if we(0) = '1' then
            di0 <= di(COL_WIDTH-1 downto 0);
        else
            di0 <= RAM(conv_integer(addr))(COL_WIDTH-1 downto 0);
        end if;
    end process;

    process(clk)
    begin
        if (clk'event and clk = '1') then
            do <= RAM(conv_integer(addr));
            RAM(conv_integer(addr)) <= di1 & di0;
        end if;
    end process;

end syn;
```

2 プロセス記述スタイルの Verilog コード例

```
//
// Single-Port BRAM with Byte-wide Write Enable (2 bytes) in Read-First Mode
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/rams/rams_24.v
//
module v_rams_24 (clk, we, addr, di, do);

    parameter SIZE      = 512;
    parameter ADDR_WIDTH = 9;
    parameter DI_WIDTH  = 8;

    input  clk;
    input  [1:0] we;
    input  [ADDR_WIDTH-1:0] addr;
    input  [2*DI_WIDTH-1:0] di;
    output [2*DI_WIDTH-1:0] do;
    reg    [2*DI_WIDTH-1:0] RAM [SIZE-1:0];
    reg    [2*DI_WIDTH-1:0] do;

    reg    [DI_WIDTH-1:0] di0, di1;

    always @(we or di)
    begin
        if (we[1])
            di1 = di[2*DI_WIDTH-1:1*DI_WIDTH];
        else
            di1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];

        if (we[0])
            di0 = di[DI_WIDTH-1:0];
        else
            di0 = RAM[addr][DI_WIDTH-1:0];
    end

    end

    always @(posedge clk)
    begin
        do <= RAM[addr];
        RAM[addr]<={di1,di0};
    end

    end

endmodule
```

非対称ポートのサポート (ブロックRAM)

ブロック RAM リソースは、2 つの非対称ポートを含めてコンフィギュレーションできます。

- ・ ポート A は特定のデータ幅で物理メモリにアクセスします。
- ・ ポート B は別のデータ幅で同じ物理メモリにアクセスします。
- ・ どちらのポートも同じ物理メモリにアクセスしますが、認識する RAM の論理構造は異なります。たとえば、同じ 2048 ビットの物理メモリは、それぞれのポートで次のように認識されます。
 - ポートA : 256x8 ビット
 - ポートB : 64x32 ビット
- ・ このように非対称にコンフィギュレーションされたブロック RAM のポートでは、アスペクト比率が異なります。
- ・ 非対称ポートは通常、ストレージの作成や 2 つのデータフロー間のバッファァーのために使用します。2 つのデータフローの特徴は、次のとおりです。
 - データ幅特性が異なる
 - 非対称速度で動作する

注記： 18 ビットを超えるデータ バスの非対称 RAM インターフェイスには、RAMB36E1 ブロックが必要になります。

非対称ポートのブロック RAM の記述

非対称ポートを含まない RAM と同様、非対称ポートを含むブロック RAM は配列オブジェクトの中の 1 つの配列を使用して記述されます。

- ・ 信号や共通変数を記述する際には、深さや幅の特性はデータ幅の小さい方 (深さの大きい方) の RAM ポートに合わせます。
- ・ この記述要件の結果、データ幅の大きい方のポートの読み出しまたは書き込みアクセスを記述する場合には、1 つの代入文ではなく、複数の代入文で記述する必要があります。
 - 代入文の数は 2 つの非対称データ幅同士の比率と同じになります。
 - これらの代入はそれぞれ次のコード例のように記述できます。

非対称ポート RAM の VHDL コード例

```
--
-- Asymmetric port RAM
--   Port A is 256x8-bit write-only
--   Port B is 64x32-bit read-only
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/asymmetric_ram_1a.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity asymmetric_ram_1a is

    generic (
        WIDTHA      : integer := 8;
        SIZEA       : integer := 256;
        ADDRWIDTHA   : integer := 8;
        WIDTHB       : integer := 32;
        SIZEB        : integer := 64;
        ADDRWIDTHB   : integer := 6
    );

    port (
        clkA      : in  std_logic;
        clkB      : in  std_logic;
        weA       : in  std_logic;
        enA       : in  std_logic;
        enB       : in  std_logic;
        addrA     : in  std_logic_vector(ADDRWIDTHA-1 downto 0);
        addrB     : in  std_logic_vector(ADDRWIDTHB-1 downto 0);
        diA       : in  std_logic_vector(WIDTHA-1 downto 0);
        doB       : out std_logic_vector(WIDTHB-1 downto 0)
    );

end asymmetric_ram_1a;

architecture behavioral of asymmetric_ram_1a is

    function max(L, R: INTEGER) return INTEGER is
    begin
        if L > R then
            return L;
        else
            return R;
        end if;
    end;

    function min(L, R: INTEGER) return INTEGER is
    begin
        if L < R then
            return L;
        else
            return R;
        end if;
    end;

    constant minWIDTH : integer := min(WIDTHA, WIDTHB);
    constant maxWIDTH : integer := max(WIDTHA, WIDTHB);
    constant maxSIZE   : integer := max(SIZEA, SIZEB);
    constant RATIO     : integer := maxWIDTH / minWIDTH;

    type ramType is array (0 to maxSIZE-1) of std_logic_vector(minWIDTH-1 downto 0);
    signal ram : ramType := (others => (others => '0'));

    signal readB : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');
    signal regB  : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');

begin
```

```
process (clkA)
begin
    if rising_edge(clkA) then
        if enA = '1' then
            if weA = '1' then
                ram(conv_integer(addrA)) <= diA;
            end if;
        end if;
    end if;
end process;

process (clkB)
begin
    if rising_edge(clkB) then
        if enB = '1' then
            readB(minWIDTH-1 downto 0)
            <= ram(conv_integer(addrB&conv_std_logic_vector(0,2)));
            readB(2*minWIDTH-1 downto minWIDTH)
            <= ram(conv_integer(addrB&conv_std_logic_vector(1,2)));
            readB(3*minWIDTH-1 downto 2*minWIDTH)
            <= ram(conv_integer(addrB&conv_std_logic_vector(2,2)));
            readB(4*minWIDTH-1 downto 3*minWIDTH)
            <= ram(conv_integer(addrB&conv_std_logic_vector(3,2)));
            end if;
            regB <= readB;
        end if;
    end process;

    doB <= regB;
end behavioral;
```

非対称ポート RAM の Verilog コード例

```
//
// Asymmetric port RAM
//   Port A is 256x8-bit write-only
//   Port B is 64x32-bit read-only
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/rams/asymmetric_ram_1a.v
//
module v_asymmetric_ram_1a (clkA, clkB, weA, reB, addrA, addrB, diA, doB);

    parameter WIDTHA      = 8;
    parameter SIZEA       = 256;
    parameter ADDRWIDTHA  = 8;
    parameter WIDTHB      = 32;
    parameter SIZEB       = 64;
    parameter ADDRWIDTHB  = 6;

    input                clkA;
    input                clkB;
    input                weA;
    input                reB;
    input    [ADDRWIDTHA-1:0]  addrA;
    input    [ADDRWIDTHB-1:0]  addrB;
    input    [WIDTHA-1:0]      diA;
    output reg  [WIDTHB-1:0]    doB;

    `define max(a,b) {(a) > (b) ? (a) : (b)}
    `define min(a,b) {(a) < (b) ? (a) : (b)}

    localparam maxSIZE = `max(SIZEA, SIZEB);
    localparam maxWIDTH = `max(WIDTHA, WIDTHB);
    localparam minWIDTH = `min(WIDTHA, WIDTHB);
    localparam RATIO    = maxWIDTH / minWIDTH;

    reg    [minWIDTH-1:0]  RAM [0:maxSIZE-1];

    reg    [WIDTHB-1:0]  readB;

    always @(posedge clkA)
    begin
        if (weA)
            RAM[addrA] <= diA;
    end

    always @(posedge clkB)
    begin
        if (reB)
            begin
                doB <= readB;
                readB[4*minWIDTH-1:3*minWIDTH] <= RAM[{addrB, 2'd3}];
                readB[3*minWIDTH-1:2*minWIDTH] <= RAM[{addrB, 2'd2}];
                readB[2*minWIDTH-1:minWIDTH] <= RAM[{addrB, 2'd1}];
                readB[minWIDTH-1:0] <= RAM[{addrB, 2'd0}];
            end
    end
endmodule
```


for-loop 文の使用

for-loop 文を使用すると、VHDL を次のようにできます。

- ・ さらにコンパクトにします
- ・ 維持しやすくします
- ・ スケールしやすくします

for-loop 文を使用した VHDL コード例

```
--
-- Asymmetric port RAM
--   Port A is 256x8-bit write-only
--   Port B is 64x32-bit read-only
--   Compact description with a for-loop statement
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/asymmetric_ram_1b.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity asymmetric_ram_1b is

    generic (
        WIDTHA      : integer := 8;
        SIZEA       : integer := 256;
        ADDRWIDTHA   : integer := 8;
        WIDTHB       : integer := 32;
        SIZEB        : integer := 64;
        ADDRWIDTHB   : integer := 6
    );

    port (
        clkA      : in  std_logic;
        clkB      : in  std_logic;
        weA       : in  std_logic;
        enA       : in  std_logic;
        enB       : in  std_logic;
        addrA     : in  std_logic_vector(ADDRWIDTHA-1 downto 0);
        addrB     : in  std_logic_vector(ADDRWIDTHB-1 downto 0);
        diA       : in  std_logic_vector(WIDTHA-1 downto 0);
        doB       : out std_logic_vector(WIDTHB-1 downto 0)
    );

end asymmetric_ram_1b;

architecture behavioral of asymmetric_ram_1b is

    function max(L, R: INTEGER) return INTEGER is
    begin
        if L > R then
            return L;
        else
            return R;
        end if;
    end;

    function min(L, R: INTEGER) return INTEGER is
    begin
        if L < R then
            return L;
        else
            return R;
        end if;
    end;

    function log2 (val: natural) return natural is
```

```

    variable res : natural;
begin
    for i in 30 downto 0 loop
        if (val >= (2**i)) then
            res := i;
            exit;
        end if;
    end loop;
    return res;
end function log2;

constant minWIDTH : integer := min(WIDTHA,WIDTHB);
constant maxWIDTH : integer := max(WIDTHA,WIDTHB);
constant maxSize : integer := max(SIZEA,SIZEB);
constant RATIO : integer := maxWIDTH / minWIDTH;

type ramType is array (0 to maxSize-1) of std_logic_vector(minWIDTH-1 downto 0);
signal ram : ramType := (others => (others => '0'));

signal readB : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');
signal regB : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');

begin

    process (clkA)
    begin
        if rising_edge(clkA) then
            if enA = '1' then
                if weA = '1' then
                    ram(conv_integer(addrA)) <= diA;
                end if;
            end if;
        end if;
    end process;

    process (clkB)
    begin
        if rising_edge(clkB) then
            if enB = '1' then
                for i in 0 to RATIO-1 loop
                    readB((i+1)*minWIDTH-1 downto i*minWIDTH)
                    <= ram(conv_integer(addrB & conv_std_logic_vector(i,log2(RATIO))));
                end loop;
            end if;
            regB <= readB;
        end if;
    end process;

    doB <= regB;

end behavioral;

```

パラメーターと generate-for 文を使用した Verilog コード例

generate-for 文を使用すると、Verilog を次のようにできます。

- ・ さらにコンパクトにします
- ・ 変更しやすくします

```

//
// Asymmetric port RAM
// Port A is 256x8-bit write-only
// Port B is 64x32-bit read-only
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/rams/v_asymmetric_ram_1b.v
//
module v_asymmetric_ram_1b (clkA, clkB, weA, reB, addrA, addrB, diA, doB);

    parameter WIDTHA      = 8;

```

```

parameter SIZEA      = 256;
parameter ADDRWIDTHA = 8;
parameter WIDTHHB    = 32;
parameter SIZEB      = 64;
parameter ADDRWIDTHB = 6;

input                clkA;
input                clkB;
input                weA;
input                reB;
input [ADDRWIDTHA-1:0] addrA;
input [ADDRWIDTHB-1:0] addrB;
input [WIDTHA-1:0]    diA;
output reg [WIDTHB-1:0] doB;

`define max(a,b) {(a) > (b) ? (a) : (b)}
`define min(a,b) {(a) < (b) ? (a) : (b)}

function integer log2;
  input integer value;
  reg [31:0] shifted;
  integer res;
begin
  if (value < 2)
    log2 = value;
  else
    begin
      shifted = value-1;
      for (res=0; shifted>0; res=res+1)
        shifted = shifted>>1;
      log2 = res;
    end
end
endfunction

localparam maxSIZE    = `max(SIZEA, SIZEB);
localparam maxWIDTH    = `max(WIDTHA, WIDTHB);
localparam minWIDTH    = `min(WIDTHA, WIDTHB);
localparam RATIO       = maxWIDTH / minWIDTH;
localparam log2RATIO   = log2(RATIO);

reg [minWIDTH-1:0] RAM [0:maxSIZE-1];

reg [WIDTHB-1:0] readB;

genvar i;

always @(posedge clkA)
begin
  if (weA)
    RAM[addrA] <= diA;
end

always @(posedge clkB)
begin
  if (reB)
    doB <= readB;
end

generate for (i = 0; i < RATIO; i = i+1)
begin: ramread
  localparam [log2RATIO-1:0] lsbaddr = i;
  always @(posedge clkB)
  begin
    readB[(i+1)*minWIDTH-1:i*minWIDTH] <= RAM[{addrB, lsbaddr}];
  end
end
endgenerate

endmodule

```

注記：これらのコード例では、min、max、log2 などの関数を使用し、コードをできるだけ一般的にし、シンプルにしています。これらの関数はデザインのどこでも定義できますが、通常は package 文で定義されます。

共通変数 (VHDL)

- ・ VHDL で「対称」ポートの RAM を記述する場合、RAM に書き込まれる2 つのポートを記述する場合にのみ共通変数が必要になります。これ以外の場合は、信号 (signal) を使用してください。
- ・ VHDL で「非対称」ポートの RAM を記述する際は、書き込みポートが 1 つしか記述されていない場合でも、共通変数 (shared variable) が必要です。この書き込みポートのデータ幅の方が大きい場合、それを記述するために複数の書き込み代入文が必要になるので、次のコード例のように共通変数 (shared variable) が必要になります。

共通変数の必要な VHDL コード例

```
--
-- Asymmetric port RAM
-- Port A is 256x8-bit read-only
-- Port B is 64x32-bit write-only
-- Compact description with a for-loop statement
-- A shared variable is necessary because of the multiple write assignments
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/asymmetric_ram_4.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity asymmetric_ram_4 is

    generic (
        WIDTHA      : integer := 8;
        SIZEA       : integer := 256;
        ADDRWIDTHA  : integer := 8;
        WIDTHB      : integer := 32;
        SIZEB       : integer := 64;
        ADDRWIDTHB  : integer := 6
    );

    port (
        clkA      : in  std_logic;
        clkB      : in  std_logic;
        reA       : in  std_logic;
        weB       : in  std_logic;
        addrA     : in  std_logic_vector(ADDRWIDTHA-1 downto 0);
        addrB     : in  std_logic_vector(ADDRWIDTHB-1 downto 0);
        diB       : in  std_logic_vector(WIDTHB-1 downto 0);
        doA       : out std_logic_vector(WIDTHA-1 downto 0)
    );

end asymmetric_ram_4;

architecture behavioral of asymmetric_ram_4 is

    function max(L, R: INTEGER) return INTEGER is
    begin
        if L > R then
            return L;
        else
            return R;
        end if;
    end;

    function min(L, R: INTEGER) return INTEGER is
    begin
        if L < R then
            return L;
        else
            return R;
        end if;
    end;
```

```

        end if;
    end;

    function log2 (val: natural) return natural is
        variable res : natural;
    begin
        for i in 30 downto 0 loop
            if (val >= (2**i)) then
                res := i;
                exit;
            end if;
        end loop;
        return res;
    end function log2;

    constant minWIDTH : integer := min(WIDTHA,WIDTHB);
    constant maxWIDTH : integer := max(WIDTHA,WIDTHB);
    constant maxSIZE : integer := max(SIZEA,SIZEB);
    constant RATIO : integer := maxWIDTH / minWIDTH;

    type ramType is array (0 to maxSIZE-1) of std_logic_vector(minWIDTH-1 downto 0);
    shared variable ram : ramType := (others => (others => '0'));

    signal readA : std_logic_vector(WIDTHA-1 downto 0) := (others => '0');
    signal regA : std_logic_vector(WIDTHA-1 downto 0) := (others => '0');

begin

    process (clkA)
    begin
        if rising_edge(clkA) then
            if reA = '1' then
                readA <= ram(conv_integer(addrA));
            end if;
            regA <= readA;
        end if;
    end process;

    process (clkB)
    begin
        if rising_edge(clkB) then
            if weB = '1' then
                for i in 0 to RATIO-1 loop
                    ram(conv_integer(addrB & conv_std_logic_vector(i,log2(RATIO))))
                    := diB((i+1)*minWIDTH-1 downto i*minWIDTH);
                end loop;
            end if;
        end if;
    end process;

    doA <= regA;

end behavioral;

```

注意： 共通変数 (shared variable) は変数の延長で、すべての基本的な特性を引き継ぐことができ、プロセス間通信を可能にします。使用する際には注意してください。

- ・ 順次プロセスで記述されている順序で、記述された機能が調整されます。
- ・ 同じシミュレーションサイクルで共通変数 (shared variable) へ2 つ以上のプロセスを代入すると、予測不可能な結果になることがあります。

読み出し/書き込み同期

- ・ 読み出し/書き込み同期は、対称 RAM でも非対称 RAM でも同じように制御されます。
- ・ 次のコード例は、2 つの非対称読み出し/書き込みポートを持つ RAM を記述し、write-first、read-first および no-change 同期の記述方法をそれぞれ示しています。

非対称ポート RAM (Write-First) のVHDL コード例

```
--
-- Asymmetric port RAM
-- Port A is 256x8-bit read-and-write (write-first synchronization)
-- Port B is 64x32-bit read-and-write (write-first synchronization)
-- Compact description with a for-loop statement
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/asymmetric_ram_2b.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity asymmetric_ram_2b is

    generic (
        WIDTHA      : integer := 8;
        SIZEA       : integer := 256;
        ADDRWIDTHA   : integer := 8;
        WIDTHB       : integer := 32;
        SIZEB        : integer := 64;
        ADDRWIDTHB   : integer := 6
    );

    port (
        clkA  : in  std_logic;
        clkB  : in  std_logic;
        enA   : in  std_logic;
        enB   : in  std_logic;
        weA   : in  std_logic;
        weB   : in  std_logic;
        addrA : in  std_logic_vector(ADDRWIDTHA-1 downto 0);
        addrB : in  std_logic_vector(ADDRWIDTHB-1 downto 0);
        diA   : in  std_logic_vector(WIDTHA-1 downto 0);
        diB   : in  std_logic_vector(WIDTHB-1 downto 0);
        doA   : out std_logic_vector(WIDTHA-1 downto 0);
        doB   : out std_logic_vector(WIDTHB-1 downto 0)
    );

end asymmetric_ram_2b;

architecture behavioral of asymmetric_ram_2b is

    function max(L, R: INTEGER) return INTEGER is
    begin
        if L > R then
            return L;
        else
            return R;
        end if;
    end;

    function min(L, R: INTEGER) return INTEGER is
    begin
        if L < R then
            return L;
        else
            return R;
        end if;
    end;

    function log2 (val: natural) return natural is
        variable res : natural;
    begin
        for i in 30 downto 0 loop
            if (val >= (2**i)) then
                res := i;
                exit;
            end if;
        end loop;
    end;

end;
```

```

        end loop;
        return res;
    end function log2;

    constant minWIDTH : integer := min(WIDTHA,WIDTHB);
    constant maxWIDTH : integer := max(WIDTHA,WIDTHB);
    constant maxSIZE   : integer := max(SIZEA,SIZEB);
    constant RATIO : integer := maxWIDTH / minWIDTH;

    type ramType is array (0 to maxSIZE-1) of std_logic_vector(minWIDTH-1 downto 0);
    shared variable ram : ramType := (others => (others => '0'));

    signal readA : std_logic_vector(WIDTHA-1 downto 0) := (others => '0');
    signal readB : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');
    signal regA  : std_logic_vector(WIDTHA-1 downto 0) := (others => '0');
    signal regB  : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');

begin

    process (clkA)
    begin
        if rising_edge(clkA) then
            if enA = '1' then
                if weA = '1' then
                    ram(conv_integer(addrA)) := diA;
                end if;
                readA <= ram(conv_integer(addrA));
            end if;
            regA <= readA;
        end if;
    end process;

    process (clkB)
    begin
        if rising_edge(clkB) then
            if enB = '1' then
                if weB = '1' then
                    for i in 0 to RATIO-1 loop
                        ram(conv_integer(addrB & conv_std_logic_vector(i,log2(RATIO))))
                        := diB((i+1)*minWIDTH-1 downto i*minWIDTH);
                    end loop;
                end if;
                for i in 0 to RATIO-1 loop
                    readB((i+1)*minWIDTH-1 downto i*minWIDTH)
                    <= ram(conv_integer(addrB & conv_std_logic_vector(i,log2(RATIO))));
                end loop;
            end if;
            regB <= readB;
        end if;
    end process;

    doA <= regA;
    doB <= regB;

end behavioral;

```


非対称ポート RAM (Read-First) のVHDL コード例

```
--
-- Asymmetric port RAM
-- Port A is 256x8-bit read-and-write (read-first synchronization)
-- Port B is 64x32-bit read-and-write (read-first synchronization)
-- Compact description with a for-loop statement
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/asymmetric_ram_2c.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity asymmetric_ram_2c is

    generic (
        WIDTHA      : integer := 8;
        SIZEA       : integer := 256;
        ADDRWIDTHA   : integer := 8;
        WIDTHB       : integer := 32;
        SIZEB        : integer := 64;
        ADDRWIDTHB   : integer := 6
    );

    port (
        clkA  : in  std_logic;
        clkB  : in  std_logic;
        enA   : in  std_logic;
        enB   : in  std_logic;
        weA   : in  std_logic;
        weB   : in  std_logic;
        addrA : in  std_logic_vector(ADDRWIDTHA-1 downto 0);
        addrB : in  std_logic_vector(ADDRWIDTHB-1 downto 0);
        diA   : in  std_logic_vector(WIDTHA-1 downto 0);
        diB   : in  std_logic_vector(WIDTHB-1 downto 0);
        doA   : out std_logic_vector(WIDTHA-1 downto 0);
        doB   : out std_logic_vector(WIDTHB-1 downto 0)
    );

end asymmetric_ram_2c;

architecture behavioral of asymmetric_ram_2c is

    function max(L, R: INTEGER) return INTEGER is
    begin
        if L > R then
            return L;
        else
            return R;
        end if;
    end;

    function min(L, R: INTEGER) return INTEGER is
    begin
        if L < R then
            return L;
        else
            return R;
        end if;
    end;

    function log2 (val: natural) return natural is
        variable res : natural;
    begin
        for i in 30 downto 0 loop
            if (val >= (2**i)) then
                res := i;
                exit;
            end if;
        end loop;
    end;

end;
```

```

        end loop;
        return res;
    end function log2;

    constant minWIDTH : integer := min(WIDTHA,WIDTHB);
    constant maxWIDTH : integer := max(WIDTHA,WIDTHB);
    constant maxSIZE   : integer := max(SIZEA,SIZEB);
    constant RATIO : integer := maxWIDTH / minWIDTH;

    type ramType is array (0 to maxSIZE-1) of std_logic_vector(minWIDTH-1 downto 0);
    shared variable ram : ramType := (others => (others => '0'));

    signal readA : std_logic_vector(WIDTHA-1 downto 0) := (others => '0');
    signal readB : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');
    signal regA  : std_logic_vector(WIDTHA-1 downto 0) := (others => '0');
    signal regB  : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');

begin

    process (clkA)
    begin
        if rising_edge(clkA) then
            if enA = '1' then
                readA <= ram(conv_integer(addrA));
                if weA = '1' then
                    ram(conv_integer(addrA)) := diA;
                end if;
            end if;
            regA <= readA;
        end if;
    end process;

    process (clkB)
    begin
        if rising_edge(clkB) then
            if enB = '1' then
                for i in 0 to RATIO-1 loop
                    readB((i+1)*minWIDTH-1 downto i*minWIDTH)
                    <= ram(conv_integer(addrB & conv_std_logic_vector(i,log2(RATIO))));
                end loop;
                if weB = '1' then
                    for i in 0 to RATIO-1 loop
                        ram(conv_integer(addrB & conv_std_logic_vector(i,log2(RATIO))))
                        := diB((i+1)*minWIDTH-1 downto i*minWIDTH);
                    end loop;
                end if;
            end if;
            regB <= readB;
        end if;
    end process;

    doA <= regA;
    doB <= regB;

end behavioral;

```

非対称ポート RAM (No-Change) のVHDL コード例

```
--
-- Asymmetric port RAM
-- Port A is 256x8-bit read-and-write (no-change synchronization)
-- Port B is 64x32-bit read-and-write (no-change synchronization)
-- Compact description with a for-loop statement
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/asymmetric_ram_2d.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity asymmetric_ram_2d is

    generic (
        WIDTHA      : integer := 8;
        SIZEA       : integer := 256;
        ADDRWIDTHA   : integer := 8;
        WIDTHB       : integer := 32;
        SIZEB        : integer := 64;
        ADDRWIDTHB   : integer := 6
    );

    port (
        clkA  : in  std_logic;
        clkB  : in  std_logic;
        enA   : in  std_logic;
        enB   : in  std_logic;
        weA   : in  std_logic;
        weB   : in  std_logic;
        addrA : in  std_logic_vector(ADDRWIDTHA-1 downto 0);
        addrB : in  std_logic_vector(ADDRWIDTHB-1 downto 0);
        diA   : in  std_logic_vector(WIDTHA-1 downto 0);
        diB   : in  std_logic_vector(WIDTHB-1 downto 0);
        doA   : out std_logic_vector(WIDTHA-1 downto 0);
        doB   : out std_logic_vector(WIDTHB-1 downto 0)
    );

end asymmetric_ram_2d;

architecture behavioral of asymmetric_ram_2d is

    function max(L, R: INTEGER) return INTEGER is
    begin
        if L > R then
            return L;
        else
            return R;
        end if;
    end;

    function min(L, R: INTEGER) return INTEGER is
    begin
        if L < R then
            return L;
        else
            return R;
        end if;
    end;

    function log2 (val: natural) return natural is
        variable res : natural;
    begin
        for i in 30 downto 0 loop
            if (val >= (2**i)) then
                res := i;
                exit;
            end if;
        end loop;
    end;

end;
```

```

        end loop;
        return res;
    end function log2;

    constant minWIDTH : integer := min(WIDTHA,WIDTHB);
    constant maxWIDTH : integer := max(WIDTHA,WIDTHB);
    constant maxSIZE   : integer := max(SIZEA,SIZEB);
    constant RATIO : integer := maxWIDTH / minWIDTH;

    type ramType is array (0 to maxSIZE-1) of std_logic_vector(minWIDTH-1 downto 0);
    shared variable ram : ramType := (others => (others => '0'));

    signal readA : std_logic_vector(WIDTHA-1 downto 0) := (others => '0');
    signal readB : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');
    signal regA  : std_logic_vector(WIDTHA-1 downto 0) := (others => '0');
    signal regB  : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');

begin

    process (clkA)
    begin
        if rising_edge(clkA) then
            if enA = '1' then
                if weA = '1' then
                    ram(conv_integer(addrA)) := diA;
                else
                    readA <= ram(conv_integer(addrA));
                end if;
            end if;
            regA <= readA;
        end if;
    end process;

    process (clkB)
    begin
        if rising_edge(clkB) then
            if enB = '1' then
                for i in 0 to RATIO-1 loop
                    if weB = '1' then
                        ram(conv_integer(addrB & conv_std_logic_vector(i,log2(RATIO))))
                        := diB((i+1)*minWIDTH-1 downto i*minWIDTH);
                    else
                        readB((i+1)*minWIDTH-1 downto i*minWIDTH)
                        <= ram(conv_integer(addrB & conv_std_logic_vector(i,log2(RATIO))));
                    end if;
                end loop;
            end if;
            regB <= readB;
        end if;
    end process;

    doA <= regA;
    doB <= regB;

end behavioral;

```

パリティビット

非対称ポート RAM の場合、XST では使用可能なブロック RAM のパリティビットの利点を生かし、ワードサイズ 9、18、36 ビット用に余分なデータビットをインプリメントできます。

非対称ポート RAM (パリティビット) の VHDL コード例

```
--
-- Asymmetric port RAM
--   Port A is 2048x18-bit write-only
--   Port B is 4096x9-bit read-only
--   XST uses parity bits to accomodate data widths
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/asymmetric_ram_3.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity asymmetric_ram_3 is

    generic (
        WIDTHA      : integer := 18;
        SIZEA       : integer := 2048;
        ADDRWIDTHA   : integer := 11;
        WIDTHB       : integer := 9;
        SIZEB        : integer := 4096;
        ADDRWIDTHB   : integer := 12
    );

    port (
        clkA   : in  std_logic;
        clkB   : in  std_logic;
        weA    : in  std_logic;
        reB    : in  std_logic;
        addrA  : in  std_logic_vector(ADDRWIDTHA-1 downto 0);
        addrB  : in  std_logic_vector(ADDRWIDTHB-1 downto 0);
        diA    : in  std_logic_vector(WIDTHA-1 downto 0);
        doB    : out std_logic_vector(WIDTHB-1 downto 0)
    );

end asymmetric_ram_3;

architecture behavioral of asymmetric_ram_3 is

    function max(L, R: INTEGER) return INTEGER is
    begin
        if L > R then
            return L;
        else
            return R;
        end if;
    end;

    function min(L, R: INTEGER) return INTEGER is
    begin
        if L < R then
            return L;
        else
            return R;
        end if;
    end;

    function log2 (val: natural) return natural is
        variable res : natural;
    begin
        for i in 30 downto 0 loop
            if (val >= (2**i)) then
                res := i;
            end if;
        end loop;
    end;
```

```

        exit;
    end if;
end loop;
return res;
end function log2;

constant minWIDTH : integer := min(WIDTHA,WIDTHB);
constant maxWIDTH : integer := max(WIDTHA,WIDTHB);
constant maxSize  : integer := max(SIZEA,SIZEB);
constant RATIO    : integer := maxWIDTH / minWIDTH;

type ramType is array (0 to maxSize-1) of std_logic_vector(minWIDTH-1 downto 0);
shared variable ram : ramType := (others => (others => '0'));

signal readB : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');
signal regB  : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');

begin

process (clkA)
begin
    if rising_edge(clkA) then
        if weA = '1' then
            for i in 0 to RATIO-1 loop
                ram(conv_integer(addrA & conv_std_logic_vector(i,log2(RATIO))))
                := diA((i+1)*minWIDTH-1 downto i*minWIDTH);
            end loop;
        end if;
    end if;
end process;

process (clkB)
begin
    if rising_edge(clkB) then
        regB <= readB;
        if reB = '1' then
            readB <= ram(conv_integer(addrB));
        end if;
    end if;
end process;

doB <= regB;

end behavioral;

```

非対称ポートのガイドライン

合成済みソリューションが専用ブロック RAM リソースに最適にインプリメントされるようにするには、次のガイドラインに従います。

- ・ 非対称ポートは、記述した RAM がブロック RAM リソースにインプリメントできる場合にのみサポートされます。適切なデータ読み出し同期を必ず指定するようにしてください。
- ・ 記述した RAM が1 つのブロック RAM プリミティブにフィットする場合にのみ非対称ポートがサポートされます。
- ・ 記述した非対称ポート RAM が1 つのブロック RAM プリミティブにフィットしない場合は、必要なデバイス プリミティブを手動でインスタンスエートする必要があります。
- ・ XST で非対称にコンフィギュレーションされたブロック RAM リソースが使用できない場合、記述した RAM は LUT リソースにインプリメントされるので、最適な結果にはならず、ランタイムもかなり増加します。
- ・ 両方のポートからアクセス可能なメモリ容量は、まったく同じである必要があります。

たとえば、RAM を 256x8 ビット (2048 ビット メモリ) と認識するポートは、もう一方のポートがそれを 64x12 ビット (768 ビット メモリ) と認識する場合は記述できません。

- ・ 両方のデータ幅の比率は 2 のべき乗になります。
- ・ 両方のポートの深さの比率は 2 のべき乗になります。

非対称ポートのレポート例

```

=====
*                               HDL Synthesis                               *
=====

Synthesizing Unit <asymmetric_ram_1a>.
  Found 256x8:64x32-bit dual-port RAM <Mram_ram> for signal <ram>.
  Found 32-bit register for signal <doB>.
  Found 32-bit register for signal <readB>.
  Summary:
    inferred 1 RAM(s).
    inferred 64 D-type flip-flop(s).
Unit <asymmetric_ram_1a> synthesized.

=====
HDL Synthesis Report

Macro Statistics
# RAMs                               : 1
  256x8:64x32-bit dual-port RAM      : 1
# Registers                           : 2
  32-bit register                     : 2

=====

=====
*                               Advanced HDL Synthesis                       *
=====

Synthesizing (advanced) Unit <asymmetric_ram_1a>.
INFO:Xst - The RAM <Mram_ram> will be implemented as a BLOCK RAM,
absorbing the following Register(s): <readB> <doB>

-----
| ram_type           | Block                               |      |
-----
| Port A             |                                     |      |
|   aspect ratio     | 256-word x 8-bit                   |      |
|   mode              | read-first                         |      |
|   clkA              | connected to signal <clkA>         | rise  |
|   weA               | connected to signal <weA_0>        | high  |
|   addrA             | connected to signal <addrA>        |      |
|   diA               | connected to signal <diA>          |      |
-----
| optimization       | speed                             |      |
-----
| Port B             |                                     |      |
|   aspect ratio     | 64-word x 32-bit                   |      |
|   mode              | write-first                        |      |
|   clkB              | connected to signal <clkB>         | rise  |
|   enB               | connected to signal <enB>          | high  |
|   addrB             | connected to signal <addrB>        |      |
|   doB               | connected to signal <doB>          |      |
-----
| optimization       | speed                             |      |
-----

Unit <asymmetric_ram_1a> synthesized (advanced).

=====
Advanced HDL Synthesis Report

Macro Statistics
# RAMs                               : 1
  256x8:64x32-bit dual-port block RAM : 1

=====
...

```


RAM の初期内容

RAM の初期内容のタスクには、次が含まれます。

- ・ [HDL ソース コードでの RAM の初期内容の指定](#)
- ・ [外部データ ファイルでの RAM の初期内容の指定](#)

HDL ソース コードでの RAM の初期内容の指定

信号のデフォルト値のメカニズムを使用して、RAM の初期内容を次のようなソース コードで直接記述します。

VHDL コード例 1

```
type ram_type is array (0 to 31) of std_logic_vector(19 downto 0);
signal RAM : ram_type :=
(
    X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A", X"00300", X"08602",
    X"02310", X"0203B", X"08300", X"04002", X"08201", X"00500", X"04001", X"02500",
    X"00340", X"00241", X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
    X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021", X"0030D", X"08201"
);
```

VHDL コード例 2

アドレス指定可能なワードがすべて同じ値に初期化されます。

```
type ram_type is array (0 to 127) of std_logic_vector (15 downto 0);
signal RAM : ram_type := (others => "0000111100110101");
```

VHDL コード例 3

ビット位置すべてがすべて同じ値に初期化されます。

```
type ram_type is array (0 to 127) of std_logic_vector (15 downto 0);
signal RAM : ram_type := (others => (others => '1'));
```

VHDL コード例 4

特定のアドレス位置や範囲に対して特定の値を選択して定義します。

```
type ram_type is array (255 downto 0) of std_logic_vector (15 downto 0);
signal RAM : ram_type:= (
    196 downto 110 => X"B8B8",
    100             => X"FEFC"
    99 downto 0     => X"8282",
    others          => X"3344");
```

Verilog コード例 1

初期ブロックを使用します。

```
reg [19:0] ram [31:0];

initial begin
    ram[31] = 20'h0200A; ram[30] = 20'h00300; ram[39] = 20'h08101;
    (...)
    ram[2] = 20'h02341; ram[1] = 20'h08201; ram[0] = 20'h0400D;
end
```

Verilog コード例 2

アドレス指定可能なワードがすべて同じ値に初期化されます。

```
Reg [DATA_WIDTH-1:0] ram [DEPTH-1:0];

integer i;
initial for (i=0; i<DEPTH; i=i+1) ram[i] = 0;
```

Verilog コード例 3

特定のアドレス位置やアドレス範囲が初期化されます。

```
reg [15:0] ram [255:0];

integer index;
initial begin
    for (index = 0 ; index <= 97 ; index = index + 1)
        ram[index] = 16'h8282;
    ram[98] <= 16'h1111;
    ram[99] <= 16'h7778;
    for (index = 100 ; index <= 255 ; index = index + 1)
        ram[index] = 16'hB8B8;
end
```

外部データ ファイルでの RAM の初期内容の指定

- ・ HDL ソース コードでファイルの読み出し関数を使用して、外部データ ファイルから RAM の初期内容を読み込みます。
 - 外部データ ファイルは ASCII 形式のテキスト ファイルで、ファイル名は何にでもできます。
 - 外部データ ファイルの各行では RAM のアドレス位置の初期内容について記述します。
 - RAM 配列の行数とこのファイルの行数は同数である必要があります。行数が足りないと、メッセージが表示されます。
 - 行に関するアドレス指定可能な位置は、RAM を記述する信号の主な範囲の方向で定義されます。
 - RAM の内容は 2 進数または 16 進数で記述できます。2 進数と 16 進数を混ざって使用することはできません。
 - 外部データ ファイルには、コメントのようなその他の内容を含めることはできません。
- ・ 次の外部データ ファイルでは、8 X 32 ビット RAM を 2 進数値で初期化しています。

```
00001111000011110000111100001111
01001010001000001100000010000100
00000000001111100000000001000001
11111101010000011100010000100100
00001111000011110000111100001111
01001010001000001100000010000100
00000000001111100000000001000001
11111101010000011100010000100100
```

- ・ 詳細は、次を参照してください。
 - [VHDL のファイル タイプ サポート](#)
 - [第 5 章「Verilog ビヘイビア言語」](#)

VHDL コード例

データを次のように読み込みます。

```
type RamType is array(0 to 127) of bit_vector(31 downto 0);

impure function InitRamFromFile (RamFileName : in string) return RamType is
    FILE RamFile : text is in RamFileName;
    variable RamFileLine : line;
    variable RAM : RamType;
begin
    for I in RamType'range loop
        readline (RamFile, RamFileLine);
        read (RamFileLine, RAM(I));
    end loop;
    return RAM;
end function;

signal RAM : RamType := InitRamFromFile("rams_20c.data");
```

Verilog コード例

2 進数データの場合は \$readmemb で、16 進数のデータの場合は \$readmemh で読み込みます。

```
reg [31:0] ram [0:63];

initial begin
    $readmemb("rams_20c.data", ram, 0, 63);
end
```

ブロック RAM の最適化ストラテジ

- ・ 推論された RAM マクロが 1 つのブロック RAM に収まりきらない場合、それをさまざまな方法で複数のブロック RAM にパーティション分割することができます。
- ・ 選択した方法によって、ブロック RAM プリミティブの数やその周りのロジックの量などが異なります。
- ・ パフォーマンス、デバイス使用率、電力などの最適化トレードオフが発生します。

ブロック RAM のパフォーマンス

- ・ デフォルトのブロック RAM インプリメンテーション方法は、パフォーマンスを重視した方法です。
- ・ 複数のブロック RAM プリミティブを必要とするような RAM サイズの場合、XST では論理的なブロック RAM プリミティブ数が最小になるようなインプリメントはされません。
- ・ 小型の RAM 複数をブロック リソースにインプリメントしても、最適なパフォーマンスにはならないことがよくあります。
- ・ ブロック RAM リソースは、かなり大きなマクロを使用すれば、小型の RAM 用に使用できます。
- ・ XST は分散リソースに小型の RAM をインプリメントして、デザイン パフォーマンスを改善しようとします。
- ・ 詳細は、「[小型の RAM コンポーネントの条件](#)」を参照してください。

ブロック RAM のデバイス使用

- ・ XST では、エリア重視のブロック RAM インプリメンテーションがサポートされていません。
- ・ エリア重視のインプリメンテーションをする場合は、CORE Generator™ を使用してください。
- ・ 詳細は、[第 8 章「FPGA の最適化」](#)を参照してください。

ブロック RAM の電力削減

ブロック RAM の電力消費は抑えることができます。

- ・ [電力削減 \(POWER\)](#) 合成オプションで指定可能な最適化セットの一部を使用します。
- ・ [RAM スタイル \(RAM_STYLE\)](#) 制約でイネーブルになります。
- ・ 同時にアクティブになるブロック RAM の数を減らすことを主な目的にしています。
- ・ 次のようなメモリ にのみ適用します。
 - 複数のブロック RAM プリミティブで分割する必要がある推論済みメモリ
 - ブロック RAM リソースのイネーブル機能を利用できる推論済みメモリ
- ・ 1 つのブロック RAM プリミティブにフィットする推論済みメモリ には、この制約を使用しても効果はありません。

追加のイネーブル ロジック

推論済みメモリをインプリメントするのにブロック RAM プリミティブ 1 つだけが同時にイネーブルになるように、追加のイネーブル ロジックが作成されます。このイネーブル ロジックは、次を目標に作成されます。

- ・ 電力削減
- ・ エリアの最適化
- ・ スピードの最適化

最適化のトレードオフ

RAM スタイル (RAM_STYLE) 制約を使用すると、次の 2 つのトレードオフが使用できます。

- ・ block_power1
- ・ block_power2

block_power1

- ・ ある程度まで電力を削減
- ・ メモリ 特性によって電力削減は最小限
- ・ パフォーマンスにはあまり影響なし
- ・ 次のようなデフォルトのブロック RAM 分割方法を使用
 - － パフォーマンス重視
 - － ブロック RAM イネーブル ロジックを追加

block_power2

- ・ かなりの電力を削減
- ・ 未使用のパフォーマンス機能がある可能性あり
- ・ 追加のスライス ロジックが含まれる可能性あり
- ・ block_power1 とは異なるブロック RAM 分割方法を使用
 - － 次の方法で推論済みメモリをインプリメントするのに必要なブロック RAM プリミティブの数を削減
 - － ブロック RAM イネーブル ロジックを挿入してアクティブなブロック RAM 数を最小限に抑える
 - － アクティブなブロック RAM からデータを読み出すために、マルチプレクサ ロジックを作成

block_power2 は、次のような場合に使用します。

- ・ 電力削減が主な目的である場合
- ・ エリアと速度の最適化はある程度落としてもかまわないという場合

block_power1 および block_power2 の比較

	block_power1	block_power2
電力削減	<ul style="list-style-type: none"> ある程度まで電力を削減 メモリ 特性によって電力削減は最小限 	かなりの電力を削減
パフォーマンス	パフォーマンスにはあまり影響なし	未使用のパフォーマンス機能がある可能性あり
ブロック RAM 分割方法	デフォルトのブロック RAM 分割方法を使用	異なるブロック RAM 分割方法を使用

小型の RAM コンポーネントの条件

- ・ XST ではブロック RAM に小型のメモリ をインプリメントしません。
- ・ ブロック RAM リソースを節約するためには、インプリメントします。
- ・ しきい値には、次の点が影響します。
 - デバイス ファミリ
 - アドレス指定可能なデータワード数 (メモリ の深さ)
 - メモリ ビットの総数 (アドレス指定可能なデータワード数 * データワード幅)
- ・ 推論された RAM は、次の表の条件が満たされると、ブロック RAM リソースにインプリメントされます。
- ・ 上記の表の条件を上書きし、ブロック リソースに小型の RAM および ROM コンポーネントを強制的にブロック リソースにインプリメントするには、[RAM スタイル \(RAM_STYLE\)](#) を使用してください。

推論された RAM のブロック RAM リソースへのインプリメント条件

デバイス	メモリ 長	深さ * 幅
Spartan®-6	>= 127 ワード	> 512 ビット
Virtex®-6	>= 127 ワード	> 512 ビット
7 series	>= 127 ワード	> 512 ビット

ブロック RAM への汎用ロジックと FSM コンポーネントのインプリメント

- ・ XST では、ブロック RAM リソースに次をインプリメントできます。
 - 汎用ロジック
 - [FSM のコンポーネント](#)
- ・ 詳細は、「[ブロック RAM へのロジックのマッピング](#)」を参照してください。

ブロック RAM リソースの管理

- ・ XST では、実際に使用可能なブロック RAM リソースを考慮して、デバイスにオーバーマップされないようにします。
 - XST では、使用可能なブロック RAM リソースがすべて使用される可能性があります。
 - **BRAM 使用率 (BRAM_UTILIZATION_RATIO)** 制約を使用すると、これらのリソースを割り当てないままの状態にしておくことができます。
- ・ 推論された RAM マクロに対して使用可能な実際のブロック RAM 数は、次の数を BRAM の使用率 (BRAM_UTILIZATION_RATIO) で論理的に定義された全体数から引いて決定されます。
 1. インスタンシエートしたブロック RAM
 2. **RAM スタイル (RAM_STYLE)** および **ROM スタイル (ROM_STYLE)** 制約を使用してブロック RAM に強制的にインプリメントされた RAM と ROM コンポーネント。XST では、その他の推論された RAM コンポーネントのブロック RAM リソースへのインプリメント前に、これらの制約が適用されます。
 3. **BRAM へのロジックのマッピング (BRAM_MAP)** を使用したロジックまたは FSM のマッピングの結果のブロック RAM
- ・ XST のブロック RAM のアロケーション方法は、ブロック インプリメンテーションの最大の推論済み RAM にも使用でき、小型の RAM がデバイスに残っていない場合はブロック リソースにインプリメントされるようにできます。
- ・ 上記の 3 例から作成されるブロック RAM の合計が使用可能なリソースを上回ってしまうと、ブロック RAM が使用されすぎてしまいます。XST は、ほとんどの場合、このような使用過多が発生しないようにします。

ブロック RAM のパッキング

- ・ XST は、小型のシングル ポート RAM をまとめて、より多くの RAM をブロック リソースにインプリメントできるようにします。
- ・ 2 つのシングル ポート RAM を 1 つのデュアル ポート RAM プリミティブにインプリメントできます。この場合、各ポートでブロック RAM の物理的に区別される部分が管理されます。
- ・ この最適化は、**自動 BRAM パッキング (AUTO_BRAM_PACKING)** で制御されますが、デフォルトではディスエーブルになっています。

分散 RAM のパイプライン

- ・ XST では分散リソースにインプリメントされた RAM をパイプライン化できます。
 - 適切なレイテンシ ステージ数が必要です。
 - パイプライン化の効果は、フリップフロップのリタイミングと同様です。
 - 結果のパフォーマンスが向上します。
- ・ パイプライン ステージを挿入するには
 1. HDL ソース コードで必要なレジスタ数を記述します。
 2. それらのレジスタを RAM の後に配置します。
 3. [RAM スタイル \(RAM_STYLE\)](#) 制約を pipe_distributed に設定します。
- ・ パイプライン中、XST では次が実行されます。
 - 動作周波数を最大にするために必要な理想的なレジスタのステージ数を計算します。
 - レジスタ ステージ数が理想の数より少ない場合は、HDL Advisor のメッセージが表示されます。このメッセージには、最適な数までにあといくつレジスタ ステージ数が必要かがレポートされます。
 - レジスタに非同期セット/リセット ロジックが含まれていると、分散 RAM はパイプライン化できません。
 - レジスタに同期リセット信号が含まれている場合は、RAM をパイプライン化できます。

RAM の関連制約

- ・ RAM に関連する制約は、次のとおりです。
 - [RAM 抽出](#)
 - [RAM スタイル](#)
 - [ROM 抽出](#)
 - [ROM スタイル](#)
 - [BRAM 使用率](#)
 - [自動 BRAM パッキング](#)
- ・ XST では、1 つのブロック RAM プリミティブにインプリメントされた推論済み RAM に [LOC](#) および [RLOC](#) 制約を使用できます。
- ・ LOC および RLOC 制約は NGC ネットリストに渡されます。

RAM のレポート

- ・ XST からは、次を含む推論された RAM に関する詳細な情報がレポートされます。
 - サイズ
 - 同期化
 - 制御信号
- ・ RAM の認識は、次の 2 つの段階から構成されています。
 1. HDL 合成
HDL ソース コードにあるメモリ 構造が認識されます。
 2. アドバンス HDL 合成
各 RAM の状況のより正確な全体図が認識され、使用可能なリソースによって、それらを分散 RAM リソースにインプリメントするか、ブロック RAM リソースにインプリメントするかが決定されます。
- ・ 推論されたブロック RAM は通常次のようにレポートされます。

RAM のレポート ログの例

```
=====
*                               HDL Synthesis                               *
=====

Synthesizing Unit <rams_27>.
  Found 16-bit register for signal <do>.
  Found 128x16-bit dual-port <RAM Mram_RAM> for signal <RAM>.
  Summary:
  inferred   1 RAM(s).
  inferred  16 D-type flip-flop(s).
Unit <rams_27> synthesized.

=====
HDL Synthesis Report

Macro Statistics
# RAMs                               : 1
  128x16-bit dual-port RAM           : 1
# Registers                           : 1
  16-bit register                     : 1

=====

*                               Advanced HDL Synthesis                               *
=====

Synthesizing (advanced) Unit <rams_27>.
INFO:Xst - The <RAM Mram_RAM> will be implemented as a BLOCK RAM,
absorbing the following register(s): <do>
-----
```

ram_type	Block		

Port A			
aspect ratio	128-word x 16-bit		
mode	read-first		
clkA	connected to signal <clk>	rise	
weA	connected to signal <we>	high	
addrA	connected to signal <waddr>		
diA	connected to signal <di>		

optimization	speed		

Port B			
aspect ratio	128-word x 16-bit		
mode	write-first		
clkB	connected to signal <clk>	rise	
enB	connected to signal <re>	high	
addrB	connected to signal <raddr>		
doB	connected to signal <do>		

optimization	speed		

Unit <rams_27> synthesized (advanced).

=====

Advanced HDL Synthesis Report

Macro Statistics

```
# RAMs                                : 1
128x16-bit dual-port block RAM        : 1
```

=====

分散 RAM のパイプライン化のレポート ログの例

分散 RAM をパイプライン化すると、アドバンス HDL 合成セクションで次のように記述されます。

Synthesizing (advanced) Unit <v_rams_22>.

Found pipelined ram on signal <n0006>:

- 1 pipeline level(s) found in a register on signal <n0006>.

Pushing register(s) into the ram macro.

INFO:Xst:2390 - HDL ADVISOR - You can improve the performance of the ram Mram_RAM by adding 1 register level(s) on output signal n0006.

Unit <v_rams_22> synthesized (advanced).

RAM のコード例

アップデート情報は、「はじめに」のコード例を参照してください。

非同期読み出し付きシングル ポート RAM (分散 RAM) の VHDL コード例

```
--
-- Single-Port RAM with Asynchronous Read (Distributed RAM)
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/rams_04.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_04 is
    port (clk : in std_logic;
          we  : in std_logic;
          a   : in std_logic_vector(5 downto 0);
          di  : in std_logic_vector(15 downto 0);
          do  : out std_logic_vector(15 downto 0));
end rams_04;

architecture syn of rams_04 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
begin

    process (clk)
    begin
        if (clk'event and clk = '1') then
            if (we = '1') then
                RAM(conv_integer(a)) <= di;
            end if;
        end if;
    end process;

    do <= RAM(conv_integer(a));

end syn;
```

非同期読み出し付きデュアル ポート RAM (分散 RAM) の Verilog コード例

```
//  
// Dual-Port RAM with Asynchronous Read (Distributed RAM)  
//  
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip  
// File: HDL_Coding_Techniques/rams/rams_09.v  
//  
module v_rams_09 (clk, we, a, dpra, di, spo, dpo);  
  
    input  clk;  
    input  we;  
    input  [5:0] a;  
    input  [5:0] dpra;  
    input  [15:0] di;  
    output [15:0] spo;  
    output [15:0] dpo;  
    reg    [15:0] ram [63:0];  
  
    always @(posedge clk) begin  
        if (we)  
            ram[a] <= di;  
    end  
  
    assign spo = ram[a];  
    assign dpo = ram[dpra];  
  
endmodule
```

Read-First モードのシングル ポート ブロック RAM の VHDL コード例

```
--
-- Single-Port Block RAM Read-First Mode
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/rams_01.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_01 is
    port (clk : in std_logic;
          we : in std_logic;
          en : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          di : in std_logic_vector(15 downto 0);
          do : out std_logic_vector(15 downto 0));
end rams_01;

architecture syn of rams_01 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM: ram_type;
begin

    process (clk)
    begin
        if clk'event and clk = '1' then
            if en = '1' then
                if we = '1' then
                    RAM(conv_integer(addr)) <= di;
                end if;
                do <= RAM(conv_integer(addr)) ;
            end if;
        end if;
    end process;

end syn;
```

Read-First モードのシングル ポート ブロック RAM の Verilog コード例

```
//  
// Single-Port Block RAM Read-First Mode  
//  
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip  
// File: HDL_Coding_Techniques/rams/rams_01.v  
//  
module v_rams_01 (clk, en, we, addr, di, do);  
  
    input  clk;  
    input  we;  
    input  en;  
    input  [5:0] addr;  
    input  [15:0] di;  
    output [15:0] do;  
    reg    [15:0] RAM [63:0];  
    reg    [15:0] do;  
  
    always @(posedge clk)  
    begin  
        if (en)  
        begin  
            if (we)  
                RAM[addr]<=di;  
            do <= RAM[addr];  
        end  
    end  
  
endmodule
```

Write-First モードのシングル ポート ブロック RAM の VHDL コード例

```
--
-- Single-Port Block RAM Write-First Mode (recommended template)
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/rams_02a.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_02a is
    port (clk : in std_logic;
          we : in std_logic;
          en : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          di : in std_logic_vector(15 downto 0);
          do : out std_logic_vector(15 downto 0));
end rams_02a;

architecture syn of rams_02a is
    type ram_type is array (63 downto 0)
        of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
begin

    process (clk)
    begin
        if clk'event and clk = '1' then
            if en = '1' then
                if we = '1' then
                    RAM(conv_integer(addr)) <= di;
                    do <= di;
                else
                    do <= RAM( conv_integer(addr));
                end if;
            end if;
        end if;
    end process;

end syn;
```


Write-First モードのシングル ポート ブロック RAM の Verilog コード例

```
//  
// Single-Port Block RAM Write-First Mode (recommended template)  
//  
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip  
// File: HDL_Coding_Techniques/rams/rams_02a.v  
//  
module v_rams_02a (clk, we, en, addr, di, do);  
  
    input  clk;  
    input  we;  
    input  en;  
    input  [5:0] addr;  
    input  [15:0] di;  
    output [15:0] do;  
    reg    [15:0] RAM [63:0];  
    reg    [15:0] do;  
  
    always @(posedge clk)  
    begin  
        if (en)  
        begin  
            if (we)  
            begin  
                RAM[addr] <= di;  
                do <= di;  
            end  
            else  
                do <= RAM[addr];  
            end  
        end  
    end  
endmodule
```

No-Change モードのシングル ポート ブロック RAM の VHDL コード例

```
--
-- Single-Port Block RAM No-Change Mode
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/rams_03.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_03 is
    port (clk : in std_logic;
          we : in std_logic;
          en : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          di : in std_logic_vector(15 downto 0);
          do : out std_logic_vector(15 downto 0));
end rams_03;

architecture syn of rams_03 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
begin

    process (clk)
    begin
        if clk'event and clk = '1' then
            if en = '1' then
                if we = '1' then
                    RAM(conv_integer(addr)) <= di;
                else
                    do <= RAM( conv_integer(addr));
                end if;
            end if;
        end if;
    end process;

end syn;
```

No-Change モードのシングル ポート ブロック RAM の Verilog コード例

```
//  
// Single-Port Block RAM No-Change Mode  
//  
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip  
// File: HDL_Coding_Techniques/rams/rams_03.v  
//  
module v_rams_03 (clk, we, en, addr, di, do);  
  
    input  clk;  
    input  we;  
    input  en;  
    input  [5:0] addr;  
    input  [15:0] di;  
    output [15:0] do;  
    reg    [15:0] RAM [63:0];  
    reg    [15:0] do;  
  
    always @(posedge clk)  
    begin  
        if (en)  
        begin  
            if (we)  
                RAM[addr] <= di;  
            else  
                do <= RAM[addr];  
            end  
        end  
    end  
  
endmodule
```

2 つの書き込みポートがあるデュアル ポート ブロック RAM の VHDL コード例

```
--  
-- Dual-Port Block RAM with Two Write Ports  
-- Correct Modelization with a Shared Variable  
--  
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip  
-- File: HDL_Coding_Techniques/rams/rams_16b.vhd  
--  
library IEEE;  
use IEEE.std_logic_1164.all;  
use IEEE.std_logic_unsigned.all;  
  
entity rams_16b is  
    port (clka : in std_logic;  
          clkb : in std_logic;  
          ena  : in std_logic;  
          enb  : in std_logic;
```

```
        wea    : in std_logic;
        web    : in std_logic;
        addra  : in std_logic_vector(6 downto 0);
        addrb  : in std_logic_vector(6 downto 0);
        dia    : in std_logic_vector(15 downto 0);
        dib    : in std_logic_vector(15 downto 0);
        doa    : out std_logic_vector(15 downto 0);
        dob    : out std_logic_vector(15 downto 0));
end rams_16b;

architecture syn of rams_16b is
    type ram_type is array (127 downto 0) of std_logic_vector(15 downto 0);
    shared variable RAM : ram_type;
begin

    process (CLKA)
    begin
        if CLKA'event and CLKA = '1' then
            if ENA = '1' then
                DOA <= RAM(conv_integer(ADDRA));
                if WEA = '1' then
                    RAM(conv_integer(ADDRA)) := DIA;
                end if;
            end if;
        end if;
    end process;

    process (CLKB)
    begin
        if CLKB'event and CLKB = '1' then
            if ENB = '1' then
                DOB <= RAM(conv_integer(ADDRB));
                if WEB = '1' then
                    RAM(conv_integer(ADDRB)) := DIB;
                end if;
            end if;
        end if;
    end process;

end syn;
```

2 つの書き込みポートがあるデュアル ポート ブロック RAM の Verilog コード例

```
//
// Dual-Port Block RAM with Two Write Ports
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/rams/rams_16.v
//
module v_rams_16 (clka,clkb,ena,enb,wea,web,addra,addrb,dia,dib,doa,dob);

    input  clka,clkb,ena,enb,wea,web;
    input  [5:0]  addra,addrb;
    input  [15:0] dia,dib;
    output [15:0] doa,dob;
    reg    [15:0] ram [63:0];
    reg    [15:0] doa,dob;

    always @(posedge clka) begin
        if (ena)
            begin
                if (wea)
                    ram[addra] <= dia;
                doa <= ram[addra];
            end
        end

    always @(posedge clkb) begin
        if (enb)
            begin
                if (web)
                    ram[addrb] <= dib;
                dob <= ram[addrb];
            end
        end

endmodule
```

リセット可能なデータ出力付きブロック RAM の VHDL コード例

```
--
-- Block RAM with Resettable Data Output
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/rams_18.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_18 is
    port (clk : in std_logic;
          en  : in std_logic;
          we  : in std_logic;
          rst : in std_logic;
          addr : in std_logic_vector(6 downto 0);
          di   : in std_logic_vector(15 downto 0);
          do   : out std_logic_vector(15 downto 0));
end rams_18;

architecture syn of rams_18 is
    type ram_type is array (127 downto 0) of std_logic_vector (15 downto 0);
    signal ram : ram_type;
begin

    process (clk)
    begin
        if clk'event and clk = '1' then
            if en = '1' then -- optional enable
                if we = '1' then -- write enable
                    ram(conv_integer(addr)) <= di;
                end if;
            end if;
            if rst = '1' then -- optional reset
                do <= (others => '0');
            else
                do <= ram(conv_integer(addr));
            end if;
        end if;
    end process;

end syn;
```

リセット可能なデータ出力付きブロック RAM の Verilog コード例

```
//
// Block RAM with Resettable Data Output
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/rams/rams_18.v
//
module v_rams_18 (clk, en, we, rst, addr, di, do);

    input  clk;
    input  en;
    input  we;
    input  rst;
    input  [6:0] addr;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] ram [127:0];
    reg    [15:0] do;

    always @(posedge clk)
    begin
        if (en) // optional enable
        begin
            if (we) // write enable
                ram[addr] <= di;

            if (rst) // optional reset
                do <= 16'b0000111100001101;
            else
                do <= ram[addr];
        end
    end

endmodule
```

オプションの出力レジスタ付き ブロック RAM の VHDL コード例

```
--
-- Block RAM with Optional Output Registers
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/rams_19.vhd
--
library IEEE;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity rams_19 is
```

```
port (clk1, clk2    : in std_logic;
      we, en1, en2  : in std_logic;
      addr1         : in std_logic_vector(5 downto 0);
      addr2         : in std_logic_vector(5 downto 0);
      di            : in std_logic_vector(15 downto 0);
      res1          : out std_logic_vector(15 downto 0);
      res2          : out std_logic_vector(15 downto 0));
end rams_19;

architecture beh of rams_19 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal ram : ram_type;
    signal do1 : std_logic_vector(15 downto 0);
    signal do2 : std_logic_vector(15 downto 0);
begin

    process (clk1)
    begin
        if rising_edge(clk1) then
            if we = '1' then
                ram(conv_integer(addr1)) <= di;
            end if;
            do1 <= ram(conv_integer(addr1));
        end if;
    end process;

    process (clk2)
    begin
        if rising_edge(clk2) then
            do2 <= ram(conv_integer(addr2));
        end if;
    end process;

    process (clk1)
    begin
        if rising_edge(clk1) then
            if en1 = '1' then
                res1 <= do1;
            end if;
        end if;
    end process;

    process (clk2)
    begin
        if rising_edge(clk2) then
            if en2 = '1' then
                res2 <= do2;
            end if;
        end if;
    end process;
```



```
end beh;
```

オプションの出力レジスタ付き ブロック RAM の Verilog コード例

```
//
// Block RAM with Optional Output Registers
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/rams/rams_19.v
//
module v_rams_19 (clk1, clk2, we, en1, en2, addr1, addr2, di, res1, res2);

    input  clk1;
    input  clk2;
    input  we, en1, en2;
    input  [6:0] addr1;
    input  [6:0] addr2;
    input  [15:0] di;
    output [15:0] res1;
    output [15:0] res2;
    reg    [15:0] res1;
    reg    [15:0] res2;
    reg    [15:0] RAM [127:0];
    reg    [15:0] do1;
    reg    [15:0] do2;

    always @(posedge clk1)
    begin
        if (we == 1'b1)
            RAM[addr1] <= di;
        do1 <= RAM[addr1];
    end

    always @(posedge clk2)
    begin
        do2 <= RAM[addr2];
    end

    always @(posedge clk1)
    begin
        if (en1 == 1'b1)
            res1 <= do1;
    end

    always @(posedge clk2)
    begin
        if (en2 == 1'b1)
            res2 <= do2;
    end

endmodule
```

ブロック RAM (シングル ポート ブロック RAM) の初期化の VHDL コード例

```
--
-- Initializing Block RAM (Single-Port Block RAM)
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/rams_20a.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_20a is
    port (clk : in std_logic;
          we : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          di : in std_logic_vector(19 downto 0);
          do : out std_logic_vector(19 downto 0));
end rams_20a;

architecture syn of rams_20a is

    type ram_type is array (63 downto 0) of std_logic_vector (19 downto 0);
    signal RAM : ram_type:= (X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A",
                             X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
                             X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
                             X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
                             X"04003", X"0241B", X"00301", X"00102", X"02122", X"02021",
                             X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
                             X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
                             X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
                             X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
                             X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
                             X"0030D", X"02341", X"08201", X"0400D");

begin

    process (clk)
    begin
        if rising_edge(clk) then
            if we = '1' then
                RAM(conv_integer(addr)) <= di;
            end if;
            do <= RAM(conv_integer(addr));
        end if;
    end process;

end syn;
```

ブロック RAM (シングル ポート ブロック RAM) の初期化の Verilog コード例

```
//
// Initializing Block RAM (Single-Port Block RAM)
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/rams/rams_20a.v
//
module v_rams_20a (clk, we, addr, di, do);
    input  clk;
    input  we;
    input  [5:0] addr;
    input  [19:0] di;
    output [19:0] do;

    reg [19:0] ram [63:0];
    reg [19:0] do;

    initial begin
        ram[63] = 20'h0200A; ram[62] = 20'h00300; ram[61] = 20'h08101;
        ram[60] = 20'h04000; ram[59] = 20'h08601; ram[58] = 20'h0233A;
        ram[57] = 20'h00300; ram[56] = 20'h08602; ram[55] = 20'h02310;
        ram[54] = 20'h0203B; ram[53] = 20'h08300; ram[52] = 20'h04002;
        ram[51] = 20'h08201; ram[50] = 20'h00500; ram[49] = 20'h04001;
        ram[48] = 20'h02500; ram[47] = 20'h00340; ram[46] = 20'h00241;
        ram[45] = 20'h04002; ram[44] = 20'h08300; ram[43] = 20'h08201;
        ram[42] = 20'h00500; ram[41] = 20'h08101; ram[40] = 20'h00602;
        ram[39] = 20'h04003; ram[38] = 20'h0241E; ram[37] = 20'h00301;
        ram[36] = 20'h00102; ram[35] = 20'h02122; ram[34] = 20'h02021;
        ram[33] = 20'h00301; ram[32] = 20'h00102; ram[31] = 20'h02222;

        ram[30] = 20'h04001; ram[29] = 20'h00342; ram[28] = 20'h0232B;
        ram[27] = 20'h00900; ram[26] = 20'h00302; ram[25] = 20'h00102;
        ram[24] = 20'h04002; ram[23] = 20'h00900; ram[22] = 20'h08201;
        ram[21] = 20'h02023; ram[20] = 20'h00303; ram[19] = 20'h02433;
        ram[18] = 20'h00301; ram[17] = 20'h04004; ram[16] = 20'h00301;
        ram[15] = 20'h00102; ram[14] = 20'h02137; ram[13] = 20'h02036;
        ram[12] = 20'h00301; ram[11] = 20'h00102; ram[10] = 20'h02237;
        ram[9]  = 20'h04004; ram[8]  = 20'h00304; ram[7]  = 20'h04040;
        ram[6]  = 20'h02500; ram[5]  = 20'h02500; ram[4]  = 20'h02500;
        ram[3]  = 20'h0030D; ram[2]  = 20'h02341; ram[1]  = 20'h08201;
        ram[0]  = 20'h0400D;
    end

    always @(posedge clk)
    begin
        if (we)
            ram[addr] <= di;
        do <= ram[addr];
    end
end
```

```
endmodule
```

外部データ ファイルからのブロック RAM の初期化の VHDL コード例

```
--
-- Initializing Block RAM from external data file
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/rams_20c.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use std.textio.all;

entity rams_20c is
    port (clk : in std_logic;
          we : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          din : in std_logic_vector(31 downto 0);
          dout : out std_logic_vector(31 downto 0));
end rams_20c;

architecture syn of rams_20c is

    type RamType is array(0 to 63) of bit_vector(31 downto 0);

    impure function InitRamFromFile (RamFileName : in string) return RamType is
        FILE RamFile : text is in RamFileName;
        variable RamFileLine : line;
        variable RAM : RamType;
    begin
        for I in RamType'range loop
            readline (RamFile, RamFileLine);
            read (RamFileLine, RAM(I));
        end loop;
        return RAM;
    end function;

    signal RAM : RamType := InitRamFromFile("rams_20c.data");

begin

    process (clk)
    begin
        if clk'event and clk = '1' then
            if we = '1' then
                RAM(conv_integer(addr)) <= to_bitvector(din);
            end if;
        end if;
    end process;
end architecture;
```

```
        dout <= to_stdlogicvector(RAM(conv_integer(addr)));
    end if;
end process;

end syn;
```

外部データ ファイルからのブロック RAM の初期化の Verilog コード例

```
//
// Initializing Block RAM from external data file
// Binary data
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/rams/rams_20c.v
//
module v_rams_20c (clk, we, addr, din, dout);
    input  clk;
    input  we;
    input  [5:0] addr;
    input  [31:0] din;
    output [31:0] dout;

    reg [31:0] ram [0:63];
    reg [31:0] dout;

    initial
    begin
        // $readmemb("rams_20c.data",ram, 0, 63);
        $readmemb("rams_20c.data",ram);
    end

    always @(posedge clk)
    begin
        if (we)
            ram[addr] <= din;
        dout <= ram[addr];
    end

endmodule
```

パイプライン化された分散 RAM の VHDL コード例

```
--
-- Pipeline distributed RAM
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/rams_22.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_22 is
    port (clk : in std_logic;
          we : in std_logic;
          addr : in std_logic_vector(8 downto 0);
          di : in std_logic_vector(3 downto 0);
          do : out std_logic_vector(3 downto 0));
end rams_22;

architecture syn of rams_22 is
    type ram_type is array (511 downto 0) of std_logic_vector (3 downto 0);
    signal RAM : ram_type;

    signal pipe_reg: std_logic_vector(3 downto 0);

    attribute ram_style: string;
    attribute ram_style of RAM: signal is "pipe_distributed";
begin

    process (clk)
    begin
        if clk'event and clk = '1' then
            if we = '1' then
                RAM(conv_integer(addr)) <= di;
            else
                pipe_reg <= RAM( conv_integer(addr));
            end if;
            do <= pipe_reg;
        end if;
    end process;

end syn;
```

パイプライン化された分散 RAM の Verilog コード例

```
//
// Pipeline distributed RAM
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/rams/rams_22.v
//
module v_rams_22 (clk, we, addr, di, do);

    input        clk;
    input        we;
    input  [8:0]  addr;
    input  [3:0]  di;
    output [3:0]  do;

    (*ram_style="pipe_distributed"*)
    reg  [3:0]  RAM [511:0];
    reg  [3:0]  do;
    reg  [3:0]  pipe_reg;

    always @(posedge clk)
    begin
        if (we)
            RAM[addr] <= di;
        else
            pipe_reg <= RAM[addr];

            do <= pipe_reg;
        end
    end

endmodule
```


ROM の HDL コード記述方法

読み出し専用メモリ (ROM) は、HDL 記述とインプリメンテーションの点では、ランダム アクセス メモリ (RAM) と類似しています。XST では、ブロック RAM リソースに適切なレジスタの付いた ROM をインプリメントできます。

ROM の詳細

ROM の詳細には、次が含まれます。

- ・ [ROM の記述](#)
- ・ [読み出しアクセスの記述](#)

ROM の記述

ROM の記述には、次が含まれます。

- ・ [外部ファイルからの ROM の読み込み](#)
- ・ [VHDL での ROM の記述](#)
- ・ [Verilog での ROM の記述](#)

外部ファイルからの ROM の読み込み

- ・ 外部データ ファイルから ROM の内容を読み込むと、次のようになります。
 - HDL ソース コードはよりコンパクトで可読性あり
 - ROM データの生成や変更に柔軟性が増す
- ・ 詳細は、「[外部データ ファイルでの RAM の初期内容の指定](#)」を参照してください。

VHDL での ROM の記述

VHDL で ROM を記述するには、次に従う必要があります。

- ・ 「signal」を使用します。
これにより、次のいずれかへの ROM のインプリメンテーションを制御できます。
 - LUT リソース
 - ブロック RAM リソース
- ・ [ROM スタイル \(ROM_STYLE\)](#) または [RAM スタイル \(RAM_STYLE\)](#) 制約を信号に付けると、ROM のインプリメンテーションを制御できます。

定数ベース宣言の VHDL コード例

```
type rom_type is array (0 to 127) of std_logic_vector (19 downto 0);
constant ROM : rom_type:= (
    X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A", X"00300", X"08602",
    X"02310", X"0203B", X"08300", X"04002", X"08201", X"00500", X"04001", X"02500",
    (...)
    X"04078", X"01110", X"02500", X"02500", X"0030D", X"02341", X"08201", X"0410D"
);
```

信号ベース宣言の VHDL コード例

```
type rom_type is array (0 to 127) of std_logic_vector (19 downto 0);
signal ROM : rom_type:= (
    X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A", X"00300", X"08602",
    X"02310", X"0203B", X"08300", X"04002", X"08201", X"00500", X"04001", X"02500",
    (...)
    X"04078", X"01110", X"02500", X"02500", X"0030D", X"02341", X"08201", X"0410D"
);
```

Verilog での ROM の記述

- ・ ROM は、Verilog では初期ブロックを使用して記述できます。
- ・ Verilog では VHDL のように、1 文で 1 配列を初期化することはできません。
- ・ 各アドレス値を必ず列挙する必要があります。

初期ブロックで記述された ROM の Verilog コード例

```
reg [15:0] rom [15:0];

initial begin
    rom[0] = 16'b00111111000000010;
    rom[1] = 16'b00000000100001001;
    rom[2] = 16'b00010000000111000;
    rom[3] = 16'b00000000000000000;
    rom[4] = 16'b11000001010011000;
    rom[5] = 16'b00000000000000000;
    rom[6] = 16'b00000000110000000;
    rom[7] = 16'b01111111111100000;
    rom[8] = 16'b0010000010001001;
    rom[9] = 16'b0101010101011000;
    rom[10] = 16'b1111111010101010;
    rom[11] = 16'b00000000000000000;
    rom[12] = 16'b1110000000001000;
    rom[13] = 16'b00000000110001010;
    rom[14] = 16'b0110011100010000;
    rom[15] = 16'b00001000100000000;
end
```

case 文を使用した ROM の Verilog コード例

ROM は、case 文や if-elseif 構文を使用して記述することもできます。

```
input      [3:0] addr
output reg [15:0] data;

always @(posedge clk) begin
    if (en)
        case (addr)
            4'b0000: data <= 16'h200A;
            4'b0001: data <= 16'h0300;
            4'b0010: data <= 16'h8101;
            4'b0011: data <= 16'h4000;
            4'b0100: data <= 16'h8601;
            4'b0101: data <= 16'h233A;
            4'b0110: data <= 16'h0300;
            4'b0111: data <= 16'h8602;
            4'b1000: data <= 16'h2222;
            4'b1001: data <= 16'h4001;
            4'b1010: data <= 16'h0342;
            4'b1011: data <= 16'h232B;
            4'b1100: data <= 16'h0900;
            4'b1101: data <= 16'h0302;
            4'b1110: data <= 16'h0102;
            4'b1111: data <= 16'h4002;
        endcase
    end
```

読み出しアクセスの記述

ROM へのアクセスは、RAM へのアクセスと同じように記述できます。

読み出しアクセスを記述した VHDL コード例

conv_integer 変換関数を定義した IEEE std_logic_unsigned パッケージを含めるように指定した場合、VHDL 構文は次のようになります。

```
signal addr : std_logic_vector(ADDR_WIDTH-1 downto 0);
do <= ROM( conv_integer(addr));
```

読み出しアクセスを記述した Verilog コード例

- ・ 初期ブロックで ROM を記述した場合 (Verilog ソース コードで記述したデータを使用するか、または外部データ ファイルから読み込んだ場合)、Verilog 構文は次のようになります。

```
do <= ROM[addr];
```

- ・ または、「[case 文を使用した ROM のコード例](#)」のように case 文を使用します。

ROM のインプリメンテーション

- ・ 適切に同期された ROM がブロック RAM リソースにインプリメントできると XST で認識された場合は、「[ブロック RAM の最適化ストラテジ](#)」で説明した原則が適用されます。
- ・ デフォルトの XST の決定条件を上書きするには、[RAM スタイル \(RAM_STYLE\)](#) ではなく、[ROM スタイル \(ROM_STYLE\)](#) を使用します。
- ・ ROM スタイルの詳細は、[第 9 章「デザイン制約」](#)を参照してください。
- ・ ROM のインプリメンテーションの詳細は、[第 8 章「FPGA の最適化」](#)を参照してください。

ROM の関連制約

[ROM スタイル](#)

ROM のレポート

次のレポートは、ROM が HDL 合成中にどのように認識されたかを示しています。ROM をブロック RAM リソースにインプリメントするかどうかは、適切な同期ができるかどうかに基づいて、アドバンス HDL 合成中に決定されます。

ROM のレポート例

```
=====
*                               HDL Synthesis                               *
=====

Synthesizing Unit <roms_signal>.
  Found 20-bit register for signal <data>.
  Found 128x20-bit ROM for signal <n0024>.
  Summary:
  inferred   1 ROM(s).
  inferred  20 D-type flip-flop(s).
Unit <roms_signal> synthesized.

=====

HDL Synthesis Report

Macro Statistics
# ROMs                               : 1
  128x20-bit ROM                     : 1
# Registers                          : 1
  20-bit register                    : 1

=====

=====
*                               Advanced HDL Synthesis                       *
=====

Synthesizing (advanced) Unit <roms_signal>.
INFO:Xst - The ROM <Mrom_ROM> will be implemented as a read-only BLOCK RAM,
```

```
absorbing the register: <data>.
INFO:Xst - The RAM <Mrom_ROM> will be implemented as BLOCK RAM
```

ram_type	Block		
Port A			
aspect ratio	128-word x 20-bit		
mode	write-first		
clkA	connected to signal <clk>	rise	
enA	connected to signal <en>	high	
weA	connected to internal node	high	
addrA	connected to signal <addr>		
diA	connected to internal node		
doA	connected to signal <data>		
optimization	speed		

```
Unit <roms_signal> synthesized (advanced).
```

Advanced HDL Synthesis Report

Macro Statistics

```
# RAMs : 1
128x20-bit single-port block RAM : 1
```

ROM のコード例

アップデート情報は、「はじめに」のコード例を参照してください。

定数を使用した ROM の VHDL コード例

```
--
-- Description of a ROM with a VHDL constant
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/roms_constant.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity roms_constant is
    port (clk : in std_logic;
          en : in std_logic;
          addr : in std_logic_vector(6 downto 0);
          data : out std_logic_vector(19 downto 0));
end roms_constant;
```

architecture syn of roms_constant is

```

type rom_type is array (0 to 127) of std_logic_vector (19 downto 0);
constant ROM : rom_type:= (
    X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A", X"00300", X"08602",
    X"02310", X"0203B", X"08300", X"04002", X"08201", X"00500", X"04001", X"02500",
    X"00340", X"00241", X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
    X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021", X"00301", X"00102",
    X"02222", X"04001", X"00342", X"0232B", X"00900", X"00302", X"00102", X"04002",
    X"00900", X"08201", X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
    X"00102", X"02137", X"02036", X"00301", X"00102", X"02237", X"04004", X"00304",
    X"04040", X"02500", X"02500", X"02500", X"0030D", X"02341", X"08201", X"0400D",
    X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A", X"00300", X"08602",
    X"02310", X"0203B", X"08300", X"04002", X"08201", X"00500", X"04001", X"02500",
    X"00340", X"00241", X"04112", X"08300", X"08201", X"00500", X"08101", X"00602",
    X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021", X"00301", X"00102",
    X"02222", X"04001", X"00342", X"0232B", X"00870", X"00302", X"00102", X"04002",
    X"00900", X"08201", X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
    X"00102", X"02137", X"FF036", X"00301", X"00102", X"10237", X"04934", X"00304",
    X"04078", X"01110", X"02500", X"02500", X"0030D", X"02341", X"08201", X"0410D"
);

begin

    process (clk)
    begin
        if (clk'event and clk = '1') then
            if (en = '1') then
                data <= ROM(conv_integer(addr));
            end if;
        end if;
    end process;

end syn;
```

ブロック RAM リソースを使用した ROM の Verilog コード例

```

//
// ROMs Using Block RAM Resources.
// Verilog code for a ROM with registered output (template 1)
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/rams/rams_21a.v
//
module v_rams_21a (clk, en, addr, data);

    input      clk;
    input      en;
    input      [5:0] addr;
    output reg [19:0] data;
```

```

always @(posedge clk) begin
    if (en)
        case(addr)
            6'b000000: data <= 20'h0200A;    6'b100000: data <= 20'h02222;
            6'b000001: data <= 20'h00300;    6'b100001: data <= 20'h04001;
            6'b000010: data <= 20'h08101;    6'b100010: data <= 20'h00342;
            6'b000011: data <= 20'h04000;    6'b100011: data <= 20'h0232B;
            6'b000100: data <= 20'h08601;    6'b100100: data <= 20'h00900;
            6'b000101: data <= 20'h0233A;    6'b100101: data <= 20'h00302;
            6'b000110: data <= 20'h00300;    6'b100110: data <= 20'h00102;
            6'b000111: data <= 20'h08602;    6'b100111: data <= 20'h04002;
            6'b001000: data <= 20'h02310;    6'b101000: data <= 20'h00900;
            6'b001001: data <= 20'h0203B;    6'b101001: data <= 20'h08201;
            6'b001010: data <= 20'h08300;    6'b101010: data <= 20'h02023;
            6'b001011: data <= 20'h04002;    6'b101011: data <= 20'h00303;
            6'b001100: data <= 20'h08201;    6'b101100: data <= 20'h02433;
            6'b001101: data <= 20'h00500;    6'b101101: data <= 20'h00301;
            6'b001110: data <= 20'h04001;    6'b101110: data <= 20'h04004;
            6'b001111: data <= 20'h02500;    6'b101111: data <= 20'h00301;
            6'b010000: data <= 20'h00340;    6'b110000: data <= 20'h00102;
            6'b010001: data <= 20'h00241;    6'b110001: data <= 20'h02137;
            6'b010010: data <= 20'h04002;    6'b110010: data <= 20'h02036;
            6'b010011: data <= 20'h08300;    6'b110011: data <= 20'h00301;
            6'b010100: data <= 20'h08201;    6'b110100: data <= 20'h00102;
            6'b010101: data <= 20'h00500;    6'b110101: data <= 20'h02237;
            6'b010110: data <= 20'h08101;    6'b110110: data <= 20'h04004;
            6'b010111: data <= 20'h00602;    6'b110111: data <= 20'h00304;
            6'b011000: data <= 20'h04003;    6'b111000: data <= 20'h04040;
            6'b011001: data <= 20'h0241E;    6'b111001: data <= 20'h02500;
            6'b011010: data <= 20'h00301;    6'b111010: data <= 20'h02500;
            6'b011011: data <= 20'h00102;    6'b111011: data <= 20'h02500;
            6'b011100: data <= 20'h02122;    6'b111100: data <= 20'h0030D;
            6'b011101: data <= 20'h02021;    6'b111101: data <= 20'h02341;
            6'b011110: data <= 20'h00301;    6'b111110: data <= 20'h08201;
            6'b011111: data <= 20'h00102;    6'b111111: data <= 20'h0400D;
        endcase
    end
endmodule

```

デュアル ポート ROM の VHDL コード例

```

--
-- A dual-port ROM
-- Implementation on LUT or BRAM controlled with a ram_style constraint
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/roms_dualport.vhd
--

```

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity roms_dualport is
    port (clk          : in std_logic;
          ena,   enb   : in std_logic;
          addra, addrb : in std_logic_vector(5 downto 0);
          dataa, datab : out std_logic_vector(19 downto 0));
end roms_dualport;

architecture behavioral of roms_dualport is

    type rom_type is array (63 downto 0) of std_logic_vector (19 downto 0);
    signal ROM : rom_type:= (X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A",
                             X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
                             X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
                             X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
                             X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
                             X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
                             X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
                             X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
                             X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
                             X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
                             X"0030D", X"02341", X"08201", X"0400D");

    -- attribute ram_style : string;
    -- attribute ram_style of ROM : signal is "distributed";

begin

    process (clk)
    begin
        if rising_edge(clk) then
            if (ena = '1') then
                dataa <= ROM(conv_integer(addra));
            end if;
        end if;
    end process;

    process (clk)
    begin
        if rising_edge(clk) then
            if (enb = '1') then
                datab <= ROM(conv_integer(addrb));
            end if;
        end if;
    end process;

end behavioral;

```


FSM のコンポーネント

- ・ XST には、次のような特徴があります。
 - 同期 FSM コンポーネント専用の推論機能
 - 最適化目標に合わせたビルトイン FSM エンコーディング ストラテジ
- ・ また、ユーザー独自のエンコーディング方法が使用できるようにもできます。
- ・ FSM 抽出は、デフォルトでイネーブルになっています。
- ・ FSM の抽出をオフにするには、[FSM 自動抽出 \(FSM_ext\)](#) を使用します。

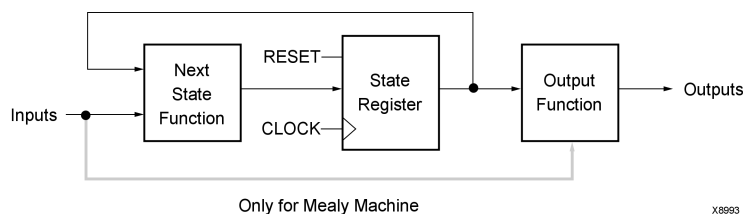
FSM の記述

- ・ XST では、ムーア型とミラー型の方の FSM (有限ステート マシン) がサポートされます。
- ・ FSM には、次が含まれます。
 - ステート レジスタ
 - 次ステート関数
 - 出力関数

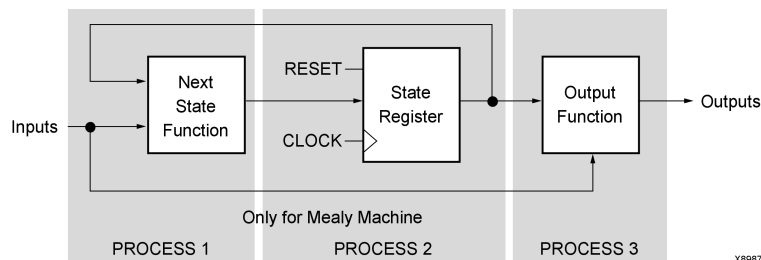
HDL コーディング手法

- ・ 目標によって、次の HDL コード記述方法のいずれかを選択できます。
- ・ 次の HDL コーディング方法を使用すると、次が可能になります。
 - 最も読みやすくなります。
 - XST で FSM を認識する機能が最大限に発揮されます。
- ・ 方法 1
FSM の 3 つすべてのコンポーネントを 1 つの順次プロセスまたは always ブロックで記述します。
- ・ 方法 2
 1. 順次プロセスまたは always ブロックでステートレジスタと次ステート関数を一緒に記述します。
 2. 別の組み合わせプロセスまたは always ブロックで出力関数を記述します。
- ・ 方法 3
 1. 順次プロセスまたは always ブロックでステートレジスタを記述します。
 2. 別の組み合わせプロセスまたは always ブロックで次ステート関数と出力関数を一緒に記述します。
- ・ 方法 4
 1. 順次プロセスまたは always ブロックでステートレジスタを記述します。
 2. 最初の組み合わせプロセスまたは always ブロックで次ステート関数を記述します。
 3. 2 つ目の別の組み合わせプロセスまたは always ブロックで出力関数を記述します。

ミーリー マシンおよびムーア マシンを取り入れた FSM の図



3つのプロセスブロックを使用した FSM の図



ステートレジスタ

- ・ リセットまたはパワーアップ ステートを指定して、FSM を認識させます。
- ・ ステートレジスタは、特定のステートに対して非同期または同期にリセットできます。
- ・ FSM の場合、非同期よりも、同期リセット ロジックをお勧めします。

VHDL でのステートレジスタの指定

VHDL では、ステートレジスタを次を使用して指定できます。

- ・ 標準型
- ・ 列挙型

標準型

ステートレジスタは、次のような標準型で指定します。

- ・ integer
- ・ bit_vector
- ・ std_logic_vector

列挙型

1. すべての可能性のあるステート値を含む列挙方を定義します。
2. ステートレジスタを列挙型を使用して宣言します。

```
type state_type is (state1, state2, state3, state4);
signal state : state_type;
```

Verilog でのステートレジスタの指定

- ・ Verilog でのステートレジスタのタイプは、次のいずれかになります。
 - 整数
 - 定義済みパラメーターのセット

```
parameter [3:0]  
    s1 = 4'b0001,  
    s2 = 4'b0010,  
    s3 = 4'b0100,  
    s4 = 4'b1000;  
reg [3:0] state;
```

- ・ これらのパラメーターは、異なるステート エンコード方法を表すよう変更できます。

次ステートの論理式

- ・ 次ステート論理式は、次のように記述できます。
 - 順次プロセスに直接、または
 - 別の組み合わせプロセスに記述
- ・ 別の組み合わせプロセスのセンシティビティリストには、次が含まれます。
 - ステート信号
 - すべての FSM 出力
- ・ 最も簡潔なコード例は、case 文を使用したものです。この場合、セレクトは現在のステート信号です。

達成不可能ステート

XST では、達成不可能な (unreachable) ステートが検出され、それについてレポートされます。

FSM の出力

- ・ レジスタの付いていない出力は、次のいずれかで記述されます。
 - 組み合わせプロセス
 - 同時処理代入文
- ・ レジスタを介する出力は、順次プロセス内で代入する必要があります。

FSM の入力

- ・ レジスタを介する入力は、内部信号を使用して記述します。
- ・ 内部信号は順次プロセスで代入されます。

ステート エンコード手法

- ・ XST には、さまざまな最適化目標や FSM パターンを適用するエンコード方法が含まれています。
- ・ ステート エンコード方法を選択するには、[FSM エンコード方法の指定 \(FSM_ENCODING\)](#) を使用します。
- ・ 詳細は、[第 9 章「デザイン制約」](#)を参照してください。

自動ステート エンコード

各 FSM に最適なエンコード アルゴリズムが自動的に選択されます。

ワンホット ステート エンコード

- ・ デフォルトのエンコーディング方法です。
- ・ 通常速度を最適にしたり、電力消費を削減する際に使用します。
- ・ 各 FSM ステートにコードの各ビットが代入されます。
- ・ 各ステートに 1 つのフリップフロップを使用したステートレジスタをインプリメントします。
 - 1 つのクロック サイクルで 1 つのステートレジスタのみがアサートされます。
 - 2 つのステート間で遷移するときには、2 つのビットのみが切り替わります。

グレイ ステート エンコード

- ・ 連続した 2 つのステート間で、1 ビットしか切り替わりません。
- ・ 分岐のない長いパスを持つコントローラに適しています。
- ・ ハザードやグリッチを最小限に抑えます。
- ・ T フリップフロップの付いたステートレジスタをインプリメントするときに使用するのをお勧めします。
- ・ 電力消費を抑えるために使用できます。

コンパクト ステート エンコード

- ・ ステート変数およびフリップフロップの数を最小限にします。ハイパーキューブ イメージョンに基づいています。
- ・ エリアを最適化する際に適しています。

ジョンソン ステート エンコード

グレイステートエンコードと同様、分岐のない長いパスを含むステートマシンに適しています。

シーケンシャル ステート エンコード

- ・ 長いパスを識別します。
- ・ これらのパスのステートに連続する基数コードを 2 つ適用します。
- ・ 次ステートの論理式を最小限に抑えます。

Speed1 ステート エンコード

- ・ スピード最適化に向いています。
- ・ ステートレジスタのビット数は、FSM によって異なりますが、通常 FSM ステート数よりも多くなります。

ユーザー ステート エンコード

ユーザーが独自のエンコード手法を HDL ファイルで指定できます。

ユーザー ステート エンコードの例

ステートレジスタが列挙型に基づいて記述されると、次のようになります。

- ・ **列挙型エンコーディング (ENUM_ENCODING)** を使用して各値に特有のバイナリ値を割り当てます。
- ・ ユーザー ステート エンコーディングを選択して XST にユーザーのコーディング方法に従うように命令します。

ブロック RAM リソースへの FSM コンポーネントのインプリメント

- ・ FSM コンポーネントはスライス ロジックにインプリメントされます。
 - スライス ロジックリソースの使用を抑えるには、FSM コンポーネントがブロック RAM にインプリメントされるように指定します。
 - FSM コンポーネントをブロック RAM にインプリメントすると、大型の FSM コンポーネントのパフォーマンスを拡張できます。
- ・ スライス ロジックのインプリメンテーションを選択するには、**FSM スタイル** 制約を使用して次のいずれかを選択します。
 - デフォルト インプリメンテーション
 - ブロック RAM インプリメンテーション
- ・ この制約に使用できる値は、次のとおりです。
 - lut (デフォルト)
 - bram
- ・ XST でブロック RAM へ FSM がインプリメントできない場合は、次のいずれかが実行されます。
 - ステート マシンがスライス ロジックにインプリメントされます。
 - アドバンス HDL 合成で警告メッセージが表示されます。
- ・ この問題は、通常 FSM に非同期リセットが含まれていると発生します。

FSM のセーフ インプリメンテーション

セーフ FSM デザインには、さまざまな方法がありますが、完璧なソリューションはありません。インプリメンテーション ストラテジを決める前に次のセクションをお読みください。

最適化

- ・ 最適化は、ほとんどのアプリケーションでよく使用されるアプローチ方法です。ほとんどのアプリケーションは標準外部条件で動作します。シングル イベント アップセット (SEU) によるこれらの一時的なエラーが原因で重要な問題が発生することはありません。
- ・ XST では、デフォルトで次が検出されて最適化されます。
 - 達成不可能なステート (論理ステートと物理ステートの両方)
 - 関連する遷移ロジック
- ・ 最適化により、ステート マシンのインプリメンテーションで次が実行されます。
 - 最小のデバイス リソースが使用される
 - 最適な回路パフォーマンスが提供される

最適化の回避

- ・ アプリケーションの中には、ソフト エラーで発生する可能性がある潜在的な悪影響を無視できない外部条件で動作するものがあります。こういったアプリケーションには、最適化は向きません。
- ・ これらのソフト エラーは、主に次のいずれかが原因で発生しています。
 - 宇宙線
 - チップ パッケージからのアルファ粒子
- ・ ステート マシンでは、ソフト エラーが特に感知されます。ステート マシンは、外部条件により不正なステートに送信されると、通常動作には戻ることができなくなります。このため、これらのエラーを検出し、リカバーできるような回路の場合、達成不可能なステートは最適化で削除されるべきではありません。
- ・ 最適化を回避するには、[セーフ インプリメンテーション](#)を使用します。XST では、ステート マシンで次が実行できるように別のロジックを作成します。
 - 不正な遷移を検出する
 - 有効なリカバリ ステートに戻る
- ・ XST では、デフォルトでリセット ステートがリカバリ ステートとして選択されます。リセット ステートが使用できない場合は、パワーアップ ステートが選択されます。[セーフ リカバリ ステート \(SAFE_RECOVERY_STATE\)](#) 制約を使用すると、手動で特定のリカバリ ステートを定義できます。

ワンホット エンコーディング VS バイナリ エンコーディング

- ・ バイナリ [ステート エンコーディング](#) 方法 (コンパクト、シーケンシャル、グレイなど) を使用すると、ステートレジスタは最小のフリップフロップ数でインプリメントされます。ワンホット エンコーディングでは、より多くのフリップフロップ (有効なステートごとに 1 つ) が使用されます。これにより、ステートレジスタに影響するシングル イベント アップセット (SEU) の発生する可能性が高くなります。
- ・ このようなマイナス面はありますが、ワンホット エンコーディングにはかなりのトポロジ的利点があります。ハミング距離が 2 なので、すべてのシングル ビット エラーが簡単に検出できます。シングル ビット エラーが原因の不正な遷移があると、ステート マシンが常に不正なステートになります。XST のセーフ インプリメンテーション ロジックを使用すると、このようなエラーが検出され、問題なくリカバーされます。
- ・ 同等のバイナリ コーディングのステート マシンのハミング距離は 1 です。この場合、シングル ビット エラーにより、ステート マシンは予測されないけれども有効なステートになります。有効なステート数が 2 のべき乗の場合、すべての可能性のあるコードの値が有効なステートに対応します。ソフト エラーは常にこのような結果を生成します。この場合、回路では不正な遷移が発生したことが検出されず、ステート マシンがその通常のステートシーケンスを超えていないかどうか検出されません。このようなランダムで制御されていないリカバリは使用できない可能性があります。

リカバリ限定ステート

- ・ ステート マシンの通常の動作ステートとは別のリカバリ ステートを定義しておくことをお勧めします。
- ・ リカバリ限定ステートを定義すると、次が実行できます。
 - ステート マシンがシングル イベント アップセット (SEU) に影響されているかどうかを検出します。
 - 通常操作に復元する前に特定の動作を実行します。特定の動作には、回路のリセットまたは回路出力へのリカバリ状態のフラグが含まれます。
- ・ 故障したステート マシンが次を実行する必要がある場合にのみ、通常の動作ステートへ直接リカバリしても問題はありません。
 - 回路の残りの部分にその一時的状況について情報を送る、または
 - ソフト エラーの後に特定の動作を実行する

FSM セーフ インプリメンテーションの VHDL コード例

```
--
-- Finite State Machine Safe Implementation VHDL Coding Example
--   One-hot encoding
--   Recovery-only state
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/state_machines/safe_fsm.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity safe_fsm is

    port(
        clk : in  std_logic;
        rst : in  std_logic;
        c   : in  std_logic_vector(3 downto 0);
        d   : in  std_logic_vector(3 downto 0);
        q   : out std_logic_vector(3 downto 0));

end safe_fsm;

architecture behavioral of safe_fsm is

    type state_t is ( idle, state0, state1, state2, recovery );
    signal state, next_state : state_t;

    attribute fsm_encoding : string;
    attribute fsm_encoding of state : signal is "one-hot";
    attribute safe_implementation : string;
    attribute safe_implementation of state : signal is "yes";
    attribute safe_recovery_state : string;
    attribute safe_recovery_state of state : signal is "recovery";
```

```
begin

process(clk)
begin
    if rising_edge(clk) then
        if rst = '1' then
            state <= idle;
        else
            state <= next_state;
        end if;
    end if;
end process;

process(state, c, d)
begin

    next_state <= state;

    case state is
        when idle =>
            if c(0) = '1' then
                next_state <= state0;
            end if;
            q <= "0000";

        when state0 =>
            if c(0) = '1' and c(1) = '1' then
                next_state <= state1;
            end if;
            q <= d;

        when state1 =>
            next_state <= state2;
            q <= "1100";

        when state2 =>
            if c(1) = '0' then
                next_state <= state1;
            elsif c(2) = '1' then
                next_state <= state2;
            elsif c(3) = '1' then
                next_state <= idle;
            end if;
            q <= "0101";

        when recovery =>
            next_state <= state0;
            q <= "1111";
```



```
end case;

end process;

end behavioral;
```

FSM セーフ インプリメンテーションの Verilog サポート

- ・ Verilog には列挙型がないので、FSM セーフ インプリメンテーションの Verilog サポートは、VHDL よりも制限があります。
- ・ 推奨：次のステート マシンのインプリメンテーションのコーディング ガイドラインに従うようにしてください。
 - 必要なエンコーディング ストラテジを手動で強制的に指定します。
 - ◆ 有効なステートそれぞれに対してコード値を明確に定義します。
 - ◆ [FSM エンコード方法の指定 \(FSM_ENCODING\)](#) を User に設定します。
 - ステート マシンの記述で読みやすいように localparam または 'define を使用し、さまざまなステートをシンボルで指定します。
 - Verilog 属性指定ではシンボルが参照できないため、リカバリ ステート値を次のいずれかにハード コードで記述します。
 - ◆ 属性文で直接文字列を指定する、または
 - ◆ 次のコード例のように 'define を使用する

FSM セーフ インプリメンテーションの Verilog コード例

```
//
// Finite State Machine Safe Implementation Verilog Coding Example
// One-hot encoding
// Recovery-only state
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/state_machines/safe_fsm.v
//
module v_safe_fsm (clk, rst, c, d, q);

    input          clk;
    input          rst;
    input [3:0]    c;
    input [3:0]    d;
    output reg [3:0] q;

    localparam [4:0]
        idle      = 5'b00001,
        state0    = 5'b00010,
        state1    = 5'b00100,
        state2    = 5'b01000,
        recovery   = 5'b10000;

    `define recovery_attr_val "10000"
```

```
(* fsm_encoding = "user",
   safe_implementation = "yes",
   safe_recovery_state = `recovery_attr_val *)
// alternatively: safe_recovery_state = "10000" *)
reg  [4:0]  state;
reg  [4:0]  next_state;

always @ (posedge clk)
begin
    if (rst)
        state <= idle;
    else
        state <= next_state;
end

always @(*)
begin

    next_state <= state;

    case (state)

        idle: begin
            if (c[0])
                next_state <= state0;
            q <= 4'b0000;
        end

        state0: begin
            if (c[0] && c[1])
                next_state <= state1;
            q <= d;
        end

        state1: begin
            next_state <= state2;
            q <= 4'b1100;
        end

        state2: begin
            if (~c[1])
                next_state <= state1;
            else
                if (c[2])
                    next_state <= state2;
                else
                    if (c[3])
                        next_state <= idle;
                    q <= 4'b0101;
                end
            end
        end
    endcase
end
```

```
end

recovery: begin
    next_state <= state0;
    q <= 4'b1111;
end

default: begin
    next_state <= recovery;
    q <= 4'b1111;
end

endcase

end

endmodule
```

FSM の関連制約

- ・ [FSM 自動抽出](#)
- ・ [FSM スタイル](#)
- ・ [FSM エンコード方法の指定](#)
- ・ [列挙型エンコード手法](#)
- ・ [セーフ インプリメンテーション](#)
- ・ [セーフ リカバリ ステート](#)

FSM のレポート

XST のログ ファイルには、FSM コンポーネントとエンコード方法について次のようにレポートされます。

FSM のレポート例

```
=====
*                               HDL Synthesis                               *
=====

Synthesizing Unit <fsm_1>.
  Found 1-bit register for signal <outp>.
  Found 2-bit register for signal <state>.
  Found finite state machine <FSM_0> for signal <state>.
  -----
  | States           | 4 |
  | Transitions     | 5 |
  | Inputs          | 1 |
  | Outputs         | 2 |
  | Clock           | clk (rising_edge) |
  | Reset           | reset (positive)  |
  | Reset type      | asynchronous       |
  | Reset State     | s1                  |
  | Power Up State  | s1                  |
  | Encoding        | gray                |
  | Implementation | LUT                  |
  -----

Summary:
inferred 1 D-type flip-flop(s).
inferred 1 Finite State Machine(s).
Unit <fsm_1> synthesized.

=====

HDL Synthesis Report

Macro Statistics
# Registers           : 1
  1-bit register      : 1
# FSMs                : 1
```

```
=====

*                               Advanced HDL Synthesis                               *
=====

Advanced HDL Synthesis Report

Macro Statistics
# FSMs                               : 1
# Registers                           : 1
  Flip-Flops                           : 1
# FSMs                               : 1

=====

*                               Low Level Synthesis                               *
=====

Optimizing FSM <state> on signal <state[1:2]> with gray encoding.
-----
State | Encoding
-----
s1    | 00
s2    | 11
s3    | 01
s4    | 10
-----
```

FSM のコード例

アップデート情報は、「はじめに」のコード例を参照してください。

1 つのプロセス文で FSM を記述した VHDL コード例

```
--
-- State Machine described with a single process
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/state_machines/state_machines_1.vhd
--
library IEEE;
use IEEE.std_logic_1164.all;

entity fsm_1 is
    port ( clk, reset, x1 : IN std_logic;
          outp           : OUT std_logic);
end entity;

architecture behavioral of fsm_1 is
    type state_type is (s1,s2,s3,s4);
    signal state : state_type ;
begin

    process (clk)
    begin
        if rising_edge(clk) then
            if (reset = '1') then
                state <= s1;
                outp <= '1';
            else
                case state is
                    when s1 => if x1='1' then
                                state <= s2;
                                outp <= '1';
                            else
                                state <= s3;
                                outp <= '0';
                            end if;
                    when s2 => state <= s4; outp <= '0';
                    when s3 => state <= s4; outp <= '0';
                    when s4 => state <= s1; outp <= '1';
                end case;
            end if;
        end if;
    end process;

end behavioral;
```

3 つの always ブロックを使用した FSM の Verilog コード例

```
//
// State Machine with three always blocks.
```

```
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/state_machines/state_machines_3.v
//
module v_fsm_3 (clk, reset, x1, outp);
    input clk, reset, x1;
    output outp;
    reg outp;
    reg [1:0] state;
    reg [1:0] next_state;

    parameter s1 = 2'b00; parameter s2 = 2'b01;
    parameter s3 = 2'b10; parameter s4 = 2'b11;

    initial begin
        state = 2'b00;
    end

    always @(posedge clk or posedge reset)
    begin
        if (reset) state <= s1;
        else state <= next_state;
    end

    always @(state or x1)
    begin
        case (state)
            s1: if (x1==1'b1)
                    next_state = s2;
                else
                    next_state = s3;
            s2: next_state = s4;
            s3: next_state = s4;
            s4: next_state = s1;
        endcase
    end

    always @(state)
    begin
        case (state)
            s1: outp = 1'b1;
            s2: outp = 1'b1;
            s3: outp = 1'b0;
            s4: outp = 1'b0;
        endcase
    end

endmodule
```

ブラック ボックス

- ・ デザインには、次で生成された EDIF または NGC ファイルを含めることができます。
 - 合成ツール
 - 回路図テキスト エディター
 - その他のデザイン入力方法
- ・ これらのモジュールを残りのデザインに関連付けるには、インスタシエートする必要があります。
 - HDL ソース コードにブラック ボックスのインスタシエーションを使用します。
 - インスタシエートされたネットリストは XST では処理されず、最終の最上位ネットリストに含まれます。
 - また、ブラック ボックスのインスタシエーションに制約を指定することも可能です。このインスタシエーションは NGC ファイルに記述されます。
- ・ デザイン ブロックの Register Transfer Level (RTL) モデルおよび EDIF ネットリストがある場合があります。
 - RTL モデルはシミュレーションにしか使用できませんが、[ボックス タイプ制約](#)を使用すると、この RTL コードを合成しないで、ブラック ボックスを作成するように設定できます。
 - EDIF ネットリストは、NGDBuild (変換) により合成されたデザインに関連付けられます。
- ・ デザインをブラック ボックスにすると、そのデザインのほかのインスタンスもブラック ボックスになります。このインスタンスに制約を指定すると、元のデザインに指定されていた制約は無視されます。
- ・ 詳細は、次を参照してください。
 - [『制約ガイド』](#)
 - [第 10 章「一般制約」](#)
 - VHDL および Verilog 言語の参照マニュアル

ブラック ボックスの関連制約

[ボックス タイプ](#)

- ・ BOX_TYPE はデバイス プリミティブをインスタシエートするために使用する制約です。
- ・ この制約を使用する前に、[「デバイス プリミティブのサポート」](#)を参照してください。

ブラック ボックスのレポート

- ・ VHDL のエラボレーション中にブラック ボックスのインスタンス化について次のようなメッセージが表示されます。

```
WARNING:HDLCompiler:89 - "example.vhd" Line 15.  <my_bbox>
remains a black-box since it has no binding entity.
```

- ・ Verilog のエラボレーション中にブラック ボックスのインスタンス化について次のようなメッセージが表示されます。

```
WARNING:HDLCompiler:1498 - "example.v" Line 27:  Empty module
<v_my_block> remains a black box.
```

- ・ ブラック ボックスが **ボックス タイプ (BOX_TYPE)** 制約を使用するように指定されている場合、XST では次が実行されます。
 - 制約値が black_box または primitive であれば、ブラック ボックスに関するメッセージが何も表示されません。
 - 制約値が user_black_box の場合、該当するエレメントのインスタンス化ごとに次のようなメッセージが表示されます。

```
Synthesizing Unit <my_top>.  Set property "box_type =
user_black_box" for instance <my_inst>.  Unit <my_top >
synthesized.
```

ブラック ボックスのコード例

アップデート情報は、「はじめに」のコード例を参照してください。

ブラック ボックスの VHDL コード例

```
--
-- Black Box
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/black_box/black_box_1.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity black_box_1 is
    port (DI_1, DI_2 : in std_logic;
          DOUT : out std_logic);
end black_box_1;

architecture archi of black_box_1 is

    component my_block
    port (I1 : in std_logic;
          I2 : in std_logic;
          O : out std_logic);
    end component;

begin

    inst: my_block port map (I1=>DI_1,I2=>DI_2,O=>DOUT);

end archi;
```

ブラック ボックスの Verilog コード例

```
//  
// Black Box  
//  
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip  
// File: HDL_Coding_Techniques/black_box/black_box_1.v  
//  
module v_my_block (in1, in2, dout);  
    input in1, in2;  
    output dout;  
endmodule  
  
module v_black_box_1 (DI_1, DI_2, DOUT);  
    input DI_1, DI_2;  
    output DOUT;  
  
    v_my_block inst (  
        .in1 (DI_1),  
        .in2 (DI_2),  
        .dout (DOUT));  
  
endmodule
```


FPGA の最適化

下位レベルの合成中、XST では次が実行されます。

1. デバイス ファミリ リソースに対して、各 VHDL エンティティまたは Verilog モジュールを別々にマップして最適化します。
2. グローバルに完了したデザインを最適化します。

詳細は、[第 12 章「FPGA 制約 \(タイミング以外\)」](#)を参照してください。

下位レベル合成の出力は、NGC ネットリスト ファイルです。

ブロック RAM へのロジックのマップ

デザインがデバイスに収まらない場合は、一部のロジックを未使用のブロック RAM に配置します。XST では、ブロック RAM に配置するロジックを自動的に判断されないので注意してください。判断させるには、XST にそう命令する必要があります。

1. RTL 記述の一部を別の階層ブロックのブロック RAM 内に配置できるように分離します。
2. HDL コードまたは XST Constraint File (XCF) ファイルのいずれかで直接、[BRAM へのロジックのマップ \(BRAM_MAP\)](#) 制約を別の階層ブロックに設定します。

ブロック RAM の条件

- ・ ブロック RAM にインプリメントされるロジックは、次の条件を満たしている必要があります。
 - すべての出力がレジスタを介するようにする。
 - ブロックに含めることのできるレジスタのレベルは 1 つで、これらは出力レジスタとする。
 - すべての出力レジスタが同じ制御信号を持つ。
 - 出力レジスタが同期リセット信号を持つ。
 - ブロックにマルチソース状況やトライステートバッファが含まれない。
 - 中間信号にKEEP 制約を使用していない。

- ・ ブロック RAM へのロジックのマッピングは、下位レベル合成段階で実行されます。問題がなくなると、次のメッセージが表示されます。

```
Entity <logic_bram_1> mapped on BRAM.
```

- ・ 上記の条件が 1 つでも満たされない場合は、ロジックはブロック RAM にマッピングされず、警告メッセージとその理由が表示されます。

```
INFO:Xst:1789 - Unable to map block <no_logic_bram> on BRAM.
```

```
Output FF <RES> must have a synchronous reset.
```

- ・ ロジックが 1 つのブロック RAM プリミティブに配置できない場合は、複数のブロック RAM が使用されます。

フリップフロップのインプリメンテーション ガイドライン

CLB フリップフロップとラッチにセットとリセットの両方がそのままインプリメントされることはありませんでした。

- ・ セットとリセットの両方を含むフリップフロップが検出されると、XST では次が実行されます。
 - セットおよびリセットのターゲットを変更
 - 別のロジックを作成
 - 非同期セットとリセットが拒否されると、次のようなエラー メッセージが表示されます。
- ・ これらの規則は、そのフリップフロップが推論されるか、古いデバイス ファミリのプリミティブ インスタンスレーションからターゲットを変更すると適用されます。

エラー メッセージの例

```
ERROR: XST:#### - This design infers one or more latches or registers with both an active asynchronous set and reset. In the Virtex®-6 and Spartan®-6 architectures this behavior creates a sub-optimal circuit in area, power and performance. To synthesis an optimal implementation it is highly recommended to either remove one set or reset or make the function synchronous. To override this error set ?retarget_active_async_set_reset option to yes.
```

レジスタの非同期設定またはリセット

レジスタを非同期に設定またはリセットするのは、ザイリンクス デバイスでサポートはされていますが、次の理由により推奨されません。

- ・ 制御セットのマッピングがやり直せなくなります。
 - ・ 次のような複数のデバイスリソースの順次機能は同期にしかセットまたはリセットできません。
 - ブロック RAM コンポーネント
 - DSP ブロック
 - ・ これらのデバイスリソースを非同期に設定またはリセットすると、次のようになります。
 - リソースを利用できなくなります。
- または
- リソースが最適にコンフィギュレーションされなくなります。

Xilinx 推奨事項

- ・ レジスタを非同期にセット/リセットしない
- ・ 同期初期化を使用します。
- ・ コーディング ガイドラインでレジスタを非同期にセットまたはリセットにする必要がある場合は、[非同期から同期へ変換 \(ASYNC_TO_SYNC\)](#) 制約を使用します。これにより、同期セット/リセットが使用できるようになります。
 - 非同期から同期への変換 (ASYNC_TO_SYNC) が影響するのは、推論済みのレジスタのみです。
 - インスタンシエートされたフリップフロップには影響しません。
- ・ セットとリセット両方が付いたフリップフロップは記述できません。
 - セットとリセットの両方を含むフリップフロップ プリミティブは同期/非同期に関わらず、使用できなくなっています。
 - XST では非同期リセットと非同期セットの両方を含むフリップフロップが拒否されます。
- ・ できる限り、セットおよびリセット ロジックを使用しないでください。たとえば、初期内容を定義して回路のグローバル リセットを使用するなど、その他のコストがより低くてすむような方法で、希望どおりの結果にできることがあります。
- ・ フリップフロップ プリミティブのクロック イネーブル、セット/リセット制御入力には常にアクティブ High で記述します。制御入力をアクティブ Low に記述すると、インバーター ロジックにより回路のパフォーマンスが悪化します。

フリップフロップのリタイミング

- ・ フリップフロップのリタイミングとは、次のためにフリップフロップおよびラッチの位置を変更する手法です。
 - 同期パスを削減
 - クロック周波数を上昇
- ・ フリップフロップのリタイミングはデフォルトでディスエーブルになっています。
- ・ デザイン ビヘイビアは変更されません。タイミング遅延のみ変更されます。

順方向および逆方向のフリップフロップのリタイミング

- ・ 順方向フリップフロップのリタイミング :
 - LUT の入力のフリップフロップのセットを出力で 1 つのフリップフロップに移動します。
 - 通常はフリップフロップの数が減ります。
- ・ 逆方向のフリップフロップのリタイミング :
 - LUT の出力の 1 つのフリップフロップを入力側のフリップフロップのセットに移動します。
 - 通常はフリップフロップの数が (場合によってはかなり) 増えます。

順方向および逆方向のフリップフロップのリタイミングのサマリ

リタイミング	フリップフロップ	位置	変更後	位置
順方向	セット	入力	1 つ	出力
逆方向	1 つ	出力	セット	入力

グローバル最適化

- ・ フリップフロップのリタイミングはグローバル最適化の一部であり、
- ・ ほかの最適化手法と同じ制約が考慮されます。
- ・ フリップフロップのリタイミングはインクリメンタルです。

リタイミングの結果挿入されたフリップフロップがタイミングを向上するために再び同じ方向 (順方向または逆方向) に移動される場合もあります。

- ・ 次のいずれかの場合、フリップフロップのリタイミングの繰り返しが停止します。
 - タイミング制約が満たされる場合
 - タイミングがそれ以上向上しない場合

フリップフロップのメッセージ

各フリップフロップが移動されると、次を示すメッセージが表示されます。

- ・ 元のフリップフロップ名と新規のフリップフロップ名
- ・ そのフリップフロップのリタイミングが順方向と逆方向のどちらであるか

フリップフロップのリタイミングの制限

フリップフロップのリタイミングには、次のような場合には実行されません。

- ・ IOB=TRUE プロパティが指定されたフリップフロップにはリタイミングは適用されません。
- ・ フリップフロップ (または出力信号) に **キープ (KEEP)** プロパティが設定されている場合は、順方向のフリップフロップのリタイミングは発生しません。
- ・ フリップフロップ (または入力信号) に **キープ (KEEP)** プロパティが設定されている場合は、逆方向のフリップフロップのリタイミングは発生しません。
- ・ インスタンス化されたフリップフロップは、**インスタンス化されたプリミティブの最適化 (OPTIMIZE_PRIMITIVES)** が yes に設定されている場合にのみ移動されます。
- ・ フリップフロップは、**インスタンス化されたプリミティブの最適化 (OPTIMIZE_PRIMITIVES)** が yes の場合にのみ、インスタンス化済みプリミティブ間で移動されます。
- ・ set と reset が付いたフリップフロップは移動されません。

フリップフロップのリタイミングの制御方法

フリップフロップのリタイミングを制御するには、次の制約を使用します。

- ・ **レジスタ自動調整**
- ・ **最初のフリップフロップ ステージの移動**
- ・ **最後のステージの移動**

エリア制約を設定した場合のスピード最適化

スライス (LUT-FF ペア) の使用率 (SLICE_UTILIZATION_RATIO) 制約には、次のような特徴があります。

- ・ マクロ推論は制御されません。
- ・ 主な目標をエリアの削減と指定した場合でも、回路全体のパフォーマンスをある程度制御できます。
- ・ デフォルトでは選択したデバイス サイズの 100% に設定されています。
- ・ 下位レベルの最適化に影響します。
 - 概算されたエリアが制約要件よりも多いと、XST ではさらにエリアを削減しようとします。
 - 概算エリアが制約要件内に収まる場合、XST ではタイミング最適化ができないかどうかチェックされ、そのソリューションがエリア制約の要件に違反していないかどうか確認されます。

下位レベルの合成のレポート例 1 (100%)

```
Found area constraint ratio of 100 (+ 5) on block tge, actual
ratio is 102. Optimizing block tge> to meet ratio 100 (+ 5) of
1536 slices Area constraint is met for block tge>, final ratio is
95.
```

このレポート例では、次が示されています。

- ・ エリア制約のターゲットは 100% に設定
- ・ 実際のデバイス使用率の最初のエリア概算は 102%
- ・ XST により最適化が実行され、95% に削減

下位レベルの合成のレポート例 2 (70%)

```
Found area constraint ratio of 70 (+ 5) on block fpga_hm, actual
ratio is 64. Optimizing block fpga_hm> to meet ratio 70 (+ 5) of
1536 slices : WARNING:Xst - Area constraint could not be met for
block tge>, final ratio is 94
```

このレポート例では、次が示されています。

- ・ エリア制約のターゲットは 70% に設定
- ・ エリア制約のターゲットは満たされていない
- ・ エリア制約のターゲットを満たすことができない場合、XST では次が実行されます。
 - タイミング最適化中にそのエリア制約を無視します。
 - 下位レベルの合成を実行して最適な周波数を達成します。
- ・ XST ではエリア制約のターゲットを満たすことができなかったため、次の警告メッセージが表示されます。

```
WARNING:Xst - Area constraint could not be met for block tge>,
final ratio is 94
```

- ・ (+5) は、エリア制約の最大マージンを示します。
 - エリア制約が満たされないと、XST では次が実行されます。
 - エリア最適化において要求されたエリアと実際のエリアとの差が 5% 以下の場合：
 - ◆ その達成されたエリアを考慮しつつタイミング最適化が実行されます。
 - ◆ 最終的なエリア ソリューションがその範囲を超えないようになります。

下位レベルの合成のレポート例 3 (55%)

```
Found area constraint ratio of 55 (+ 5) on block fpga_hm, actual
ratio is 64. Optimizing block fpga_hm> to meet ratio 55 (+ 5) of
1536 slices : Area constraint is met for block fpga_hm>, final
ratio is 60.
```

このレポート例では、次が示されています。

- ・ エリア制約のターゲットは 55% に設定
- ・ XST では 60% 達成
- ・ エリア最適化において要求されたエリアと実際のエリアとの差が 5% 以下の場合：
 - XST ではエリア制約が満たされたと判断されます。
 - それ以降の最適化でもそれが維持されると判断されます。

自動リソース管理機能の無効化

- ・ 自動的にリソースが管理されないようにするには、[スライス \(LUT-FF ペア\) 使用率 \(SLICE_UTILIZATION_RATIO\)](#) に -1 を指定します。
- ・ [スライス \(LUT-FF ペア\) 使用率 \(SLICE_UTILIZATION_RATIO\)](#) は特定のブロックに適用できます。
- ・ 次のいずれかを指定できます。
 - スライスまたは LUT-FF ペアの絶対数
 - デバイスで使用可能なリソース総計のパーセンテージ

インプリメンテーション制約

- ・ XST は次のすべてのインプリメンテーション制約を NGC 出力ファイルに記述します。
 - HDL ソース コード
 - XCF (ザイリンクス制約ファイル)
- ・ XST は、次のいずれかのためにバッファ挿入中に [KEEP](#) プロパティを生成します。
 - 最大ファンアウト制御
 - 最適化

デバイス プリミティブのサポート

XST では、ザイリンクス デバイス プリミティブを HDL ソース コードに直接インスタンスシートできます。

- ・ インスタンスシートされたプリミティブには、次のような特徴があります。
 - UNISIM ライブラリでコンパイル済み
 - XST では自動的に最適化または変更されない
 - XST で保護され、最終の NGC ネットリストで使用可能になる
- ・ [インスタンスシートされたプリミティブの最適化 \(OPTIMIZE_PRIMITIVES\)](#) 制約を使用すると、XST は残りのデザインと共にインスタンスシートされたプリミティブを最適化しようとします。ほとんどのプリミティブにタイミング情報が含まれるので、XST で効果的なタイミングドリブンの最適化が実行できるようになります。
- ・ XST では、RAM のような複雑なプリミティブのインスタンスエーションを簡単にするために、UniMacro ライブラリもサポートされています。

詳細は、[『ライブラリ ガイド』](#)を参照してください。

属性を使用したプリミティブの生成

属性により生成できるプリミティブもあります。

バッファ タイプ

特定タイプのバッファを使用するには、[バッファ タイプ \(BUFFER_TYPE\)](#) を次のいずれかに適用します。

- ・ 回路のプライマリ I/O
- ・ 内部信号

バッファ タイプ制約を使用すると、バッファが挿入されないようにできます。

I/O 規格

[I/O 規格 \(IOSTANDARD\)](#) 制約を使用すると、I/O プリミティブに I/O 規格を割り当てることができます。

次の例では、I/O ポートに PCI33_5 I/O 規格を指定しています。

```
// synthesis attribute IOSTANDARD of in1 is PCI33_5
```

プリミティブとブラック ボックス

プリミティブのサポートは、ブラック ボックスの概念に基づいています。

ブラック ボックスの詳細は、[「FSM のセーフ インプリメンテーション」](#)を参照してください。

プリミティブおよびブラック ボックスの例

この例では、ブラック ボックスのサポートとプリミティブの違いを示しています。

- ・ たとえば、デザインに MUXF5 というサブモジュールが含まれているとします。MUXF5 は、ユーザーのファンクション ブロックである場合とザイリンクス デバイス プリミティブである場合とがあります。
- ・ XST でのこのモジュールの処理において混乱が生じないようにするため、**ボックス タイプ (BOX_TYPE)** 制約を MUXF5 のコンポーネント宣言に設定する必要があります。
- ・ BOX_TYPE を MUXF5 に設定する場合、次の値を使用します。
 - **primitive** または **black_box**
そのモジュールはザイリンクス デバイス プリミティブとして処理され、クリティカル パスの概算などにこのプリミティブのパラメーターが使用されます。
 - **user_black_box**
モジュールはブラック ボックスとして処理されます。
- ・ user_black_box の名前とザイリンクス デバイス プリミティブの名前が同じ場合は、XST で次が実行されます。
 - 固有の名前に変更
 - 警告メッセージを表示
たとえば MUX5 が MUX51 に変更されると、次のような警告メッセージが表示されます。

`WARNING:Xst:79 - Model 'muxf5' has different characteristics in destination library WARNING:Xst:80 - Model name has been changed to 'muxf51'`
- ・ **ボックス タイプ (BOX_TYPE)** が MUXF5 に設定されていない場合、このブロックはユーザー階層ブロックとして処理されます。
- ・ user_black_box の名前とザイリンクス デバイス プリミティブの名前が同じ場合は、XST で次が実行されます。
 - 固有の名前に変更
 - 警告メッセージを表示

デバイス プリミティブのライブラリ

VHDL および Verilog のライブラリを使用すると、ザイリンクス デバイス プリミティブのインスタンス化が簡素化できます。

- ・ これらのライブラリにはザイリンクス デバイス プリミティブの宣言がすべて含まれます。
- ・ 各コンポーネントに **ボックス タイプ (BOX_TYPE)** 制約が設定されています。
- ・ これらのライブラリを含めた場合は、**ボックス タイプ (BOX_TYPE)** をユーザーが適用する必要はありません。

VHDL のデバイス プリミティブのライブラリ

- ・ HDL ソース コードでパッケージ vcomponents を使用して UNISIM ライブラリを宣言します。

```
library unisim;  
use unisim.vcomponents.all;
```

- ・ HDL ソース コードは、次の XST インストール ディレクトリに含まれています。

```
vhdl\src\ unisims\unisims_vcomp.vhd
```

Verilog のデバイス ライブラリ

Verilog の場合、UNISIM ライブラリがあらかじめコンパイルされています。XST では、このライブラリがデザインにリンクされます。

デバイス プリミティブのインスタンス化

デバイス プリミティブをインスタンス化するには、ジェネリック (VHDL) およびパラメーター (Verilog) に大文字を使用してください。

デバイス プリミティブのインスタンス化例

ODDR エlement は UNISIM ライブラリで次のように宣言されています。

```
component ODDR  
  generic (  
    DDR_CLK_EDGE : string := "OPPOSITE_EDGE";  
    INIT : bit := '0';  
    SRTYPE : string := "SYNC");  
  port (  
    Q : out std_ulogic;  
    C : in std_ulogic;  
    CE : in std_ulogic;  
    D1 : in std_ulogic;  
    D2 : in std_ulogic;  
    R : in std_ulogic;  
    S : in std_ulogic);  
end component;
```

- ・ このプリミティブをインスタンス化する場合、DDR_CLK_EDGE および SRTYPE の値は大文字にする必要があります。
- ・ 大文字にしないと、XST で不明の値が使用されていることを示す警告メッセージが表示されます。

INIT の使用

LUT1 のようなプリミティブでは、インスタンス化で INIT を使用できます。

INIT を最終ネットリストに渡すには、次の 2 つの方法があります。

- ・ INIT 属性をインスタンス化したプリミティブに設定します。
- ・ INIT を次のいずれかを使用して渡します。
 - ジェネリック (VHDL)
 - パラメーター (Verilog)

INIT を最終ネットリストに渡すと、合成とシミュレーションに同じコードを使用できます。

プリミティブ プロパティの指定

インスタンス化された LUT の INIT のように、VHDL のジェネリックまたは Verilog のパラメーターを使用してインスタンス化されたプリミティブにプロパティを指定します。

- ・ インスタンス化済みプリミティブのプロパティのデフォルト値は、VHDL ジェネリックまたは Verilog パラメーターから上書きできます。ほかの方法を使用すると、次のようなエラー メッセージが表示されます。

```
ERROR:Xst:3003 - "example.vhd". Line 77. Unable to set
attribute "A_INPUT" with value "CASCADE" on instance <idsp>
of block <DSP48E1>. This property is already defined with
value "DIRECT" on the block definition by a VHDL generic or a
Verilog parameter. Apply the desired value by overriding the
default VHDL generic or Verilog parameter. Using an attribute
is not allowed.
```

- ・ シミュレーション ツールではジェネリックおよびパラメーターが認識され、回路の有効化プロセスを簡素化します。

LUT2 プリミティブの INIT プロパティを設定する VHDL コード例

```
--
-- Instantiating a LUT2 primitive
-- Configured via the generics mechanism (recommended)
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: FPGA_Optimization/primitive_support/primitive_2.vhd
--
library ieee;
use ieee.std_logic_1164.all;

library unisim;
use unisim.vcomponents.all;

entity primitive_2 is
    port(I0,I1 : in std_logic;
          O    : out std_logic);
end primitive_2;

architecture beh of primitive_2 is
begin

    inst : LUT2
        generic map (INIT=>"1")
        port map (I0=>I0, I1=>I1, O=>O);

end beh;
```

LUT2 プリミティブの INIT プロパティを設定する Verilog コード例

```
//
// Instantiating a LUT2 primitive
// Configured via the parameter mechanism
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: FPGA_Optimization/primitive_support/primitive_2.v
//
module v_primitive_2 (I0,I1,O);
    input I0,I1;
    output O;

    LUT2 #(4'h1) inst (.I0(I0), .I1(I1), .O(O));

endmodule
```

プリミティブのレポート

ボックス タイプ (BOX_TYPE) とその値 (primitive) は UNISIM ライブラリの各プリミティブに適用されているので、何の警告もなく処理されます。

XST の警告

XST では、次の 2 つの条件のいずれかが発生すると警告メッセージが表示されます。

1 つ目の警告条件

- ・ ブロック (プリミティブ以外) をインスタンスエート
および
- ・ そのブロックに内容がない場合 (ロジック記述なし)

2 つ目の警告条件

- ・ ブロックにロジック記述がある
および
- ・ [ボックス タイプ \(BOX_TYPE\)](#) を user_black_box で適用した場合

警告の例

```
Elaborating entity <example> (architecture <archi>) from  
library <work>. WARNING:HDLCompiler:89 - "example.vhd" Line 15:  
<my_block> remains a black-box since it has no binding entity.
```

プリミティブの関連制約

- ・ [ボックス タイプ](#)
- ・ XST で特別な処理をせずに HDL ソース コードから NGC ファイルに渡される配置配線の制約

プリミティブのコード例

アップデート情報は、[「はじめに」](#)のコード例を参照してください。

LUT2 プリミティブをジェネリックを使用してインスタンシエートおよびコンフィギュレーションした VHDL コード例

```
--
-- Instantiating a LUT2 primitive
-- Configured via the generics mechanism (recommended)
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: FPGA_Optimization/primitive_support/primitive_2.vhd
--
library ieee;
use ieee.std_logic_1164.all;

library unisim;
use unisim.vcomponents.all;

entity primitive_2 is
    port(I0,I1 : in std_logic;
          O    : out std_logic);
end primitive_2;

architecture beh of primitive_2 is
begin

    inst : LUT2
        generic map (INIT=>"1")
        port map (I0=>I0, I1=>I1, O=>O);

end beh;
```

LUT2 プリミティブをパラメーターを使用してインスタンシエートおよびコンフィギュレーションした Verilog コード例

```
//
// Instantiating a LUT2 primitive
// Configured via the parameter mechanism
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: FPGA_Optimization/primitive_support/primitive_2.v
//
module v_primitive_2 (I0,I1,O);
    input I0,I1;
    output O;

    LUT2 #(4'h1) inst (.I0(I0), .I1(I1), .O(O));

endmodule
```

LUT2 プリミティブを Defparam を使用してインスタンス化およびコンフィギュレーションした Verilog コード例

```
//  
// Instantiating a LUT2 primitive  
// Configured via the defparam mechanism  
//  
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip  
// File: FPGA_Optimization/primitive_support/primitive_3.v  
//  
module v_primitive_3 (I0,I1,O);  
    input I0,I1;  
    output O;  
  
    LUT2 inst (.I0(I0), .I1(I1), .O(O));  
    defparam inst.INIT = 4'h1;  
  
endmodule
```

UniMacro ライブラリの使用

- ・ XST では、UniMacro というライブラリがサポートされます。
- ・ UniMacro ライブラリを使用すると、RAM のような複雑なプリミティブのインスタンス化が簡単になります。
- ・ 詳細は、『[ライブラリ ガイド](#)』を参照してください。

VHDL での UniMacro ライブラリの使用

- ・ パッケージ vcomponents を使用して unimacro ライブラリを宣言します。

```
library unimacro;  
use unimacro.vcomponents.all;
```
- ・ このパッケージの HDL ソース コードは、次のザイリンクス ソフトウェアのインストール ディレクトリに含まれます。

```
vhdl\src\unisims\unisims_vcomp.vhd
```

Verilog での UniMacro ライブラリの使用

- ・ UniMacro ライブラリがあらかじめコンパイルされています。
- ・ UniMacro ライブラリは自動的にデザインにリンクされます。

コアの処理

コアの処理には、次が含まれます。

- ・ [コアの読み込み](#)
- ・ [コアの検索](#)
- ・ [コアのレポート](#)

コアの読み込み

- ・ XST では、EDIF または NGC ネットリスト ファイル形式のコアを読み込んで、さらに正確な次を達成できます。
 - タイミングの概算
 - リソース使用率の制御
- ・ 値に optimize を使用すると、XST では次が実行できます。
 - コアのネットリストをデザイン全体にマージする
 - 最適化しようとする
- ・ コアの読み込みをイネーブルまたはディスエーブルにするには、次のいずれかの方法を使用します。
 - ISE® Design Suite
[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Read Cores]
 - [コマンドライン モード](#)
`-read_cores`

コアの検索

XST では、コアが ISE® Design Suite のプロジェクト ディレクトリから自動的に検索されます。

- ・ これ以外の場所にコアがある場合は、次のようにそのパスを指定します。
 - ISE Design Suite
[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Core Search Directories]
 - [コマンドライン モード](#)
[コアの検索ディレクトリ](#)
 - ・ 次の事項に従うことをお勧めします。
 - コアがあるディレクトリを系統的に指定
 - この情報を最新に保つ
 - ・ これらの推奨事項に従うと、次が可能になります。
 - より良いタイミングやリソース概算が算出される
 - 予測のつかないビヘイビアやデバッグしにくい状況などを避ける
- たとえば、読み込まれていないコアの内容がわからない場合 (ブラック ボックスのように見える場合)、XST ではそのコアまでのパスへのバッファ挿入が決定されにくくなるので、タイミング クロージャに悪影響を及ぼすことがあります。

コアのレポート

コアのレポート例

```
Launcher:  Executing edif2ngd -noa "my_add.edn" "my_add.ngo"
INFO:NgdBuild - Release 11.2 - edif2ngd INFO:NgdBuild - Copyright
(c) 1995-2010 Xilinx, Inc.  All rights reserved.  Writing the
design to "my_add.ngo"...  Loading core <my_add> for timing and
area information for instance <inst>.
```

LUT へのロジックのマッピング

LUT コンポーネントを直接 HDL ソース コードにインスタンス化するには、UNISIM ライブラリを使用します。

- ・ LUT のファンクションを指定するには、LUT のインスタンスに INIT 制約を設定します。
- ・ インスタンス化した LUT またはレジスタをチップの特定スライスに配置する場合は、同じインスタンスに **RLOC** 制約を設定します。
- ・ INIT でファンクションを定義するのが不都合な場合は、別の方法を使用できます。
 1. 1 つの LUT にマッピングするファンクションを HDL ソース コードの別ブロックで記述します。
 2. このブロックに **単一 LUT へのエンティティのマッピング (LUT_MAP)** 制約を設定すると、このブロックが 1 つの LUT にマッピングされます。
 3. LUT の INIT 値は XST により算出され、最適化中この LUT が保持されます。
- ・ XST では、Synplify の xc_map 属性が認識されます。
- ・ ファンクションが 1 つの LUT にマッピングできない場合、次のようなエラー メッセージが表示されます。

```
ERROR:Xst:1349 - Failed to map xcmapped entity <v_and_one> in one lut.
```

LUT へのロジック マップの Verilog コード例

次の例では、top ブロックが 2 つの AND ゲートをインスタンス化しています。

- ・ AND ゲートは and_one および and_two ブロックで記述されます。
- ・ XST では 2 つの LUT2 が生成され、1 つに結合されることはありません。

```
//
// Mapping of Logic to LUTs with the LUT_MAP constraint
// Mapped to 2 distinct LUT2s
// Mapped to 1 single LUT3 if LUT_MAP constraints are removed
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: FPGA_Optimization/lut_mapping/lut_map_1.v
//

(* LUT_MAP="yes" *)
module v_and_one (A, B, REZ);
    input A, B;
    output REZ;

    and and_inst(REZ, A, B);
endmodule

// -----

(* LUT_MAP="yes" *)
module v_and_two (A, B, REZ);
    input A, B;
    output REZ;

    or or_inst(REZ, A, B);
endmodule

// -----

module v_lut_map_1 (A, B, C, REZ);
    input A, B, C;
    output REZ;

    wire tmp;

    v_and_one inst_and_one (A, B, tmp);
    v_and_two inst_and_two (tmp, C, REZ);
endmodule
```

デバイス上の配置の制御

- ・ 次の推論されたマクロは、デバイスの特定箇所に指定して配置できます。
 - － レジスタ
 - － ブロック RAM コンポーネント
- ・ マクロの配置を制御するには、レジスタまたはブロック RAM を表す信号に **RLOC** 制約を適用します。下記のコード例を参照してください。
- ・ RLOC がレジスタに適用されると、XST で次が実行されます。
 - － RLOC が各フリップフロップに分配されます。
 - － RLOC 制約を最終ネットリストに含めます。
- ・ RLOC は、1 つのブロック RAM プリミティブにインプリメント可能な推論された RAM でサポートされています。

4 ビット レジスタの RLOC 制約の VHDL コード例

次のコード例では、4 ビット レジスタに RLOC 制約を指定しています。

```
--
-- Specification of INIT and RLOC values for a flip-flop, described at RTL level
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: FPGA_Optimization/inits_and_rlocs/inits_rlocs_3.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity inits_rlocs_3 is
    port (CLK : in std_logic;
          DI : in std_logic_vector(3 downto 0);
          DO : out std_logic_vector(3 downto 0));
end inits_rlocs_3;

architecture beh of inits_rlocs_3 is
    signal tmp: std_logic_vector(3 downto 0):="1011";

    attribute RLOC: string;
    attribute RLOC of tmp: signal is "X3Y0 X2Y0 X1Y0 X0Y0";
begin

    process (CLK)
    begin
        if (clk'event and clk='1') then
            tmp <= DI;
        end if;
    end process;

    DO <= tmp;

end beh;
```

バッファの挿入

- ・ XST では、自動的にクロックおよび I/O バッファが挿入されます。
- ・ [I/O バッファの追加 \(-iobuf\)](#) を使用すると、I/O バッファの自動挿入を有効または無効にできます。
- ・ デフォルトで、バッファ挿入は有効になっています。
- ・ クロックと I/O バッファは手動でインスタンスシートできます。
- ・ XST ではインスタンスシートされたデバイス プリミティブを変更せず、それらを最終ネットリストに渡します。

XST での PCI フローの使用

次のガイドラインに従うと、XST を使用した PCI™ フローで配置制約およびタイミング制約をすべて満たすには、次のオプションを設定できます。

1. 生成されたネットリスト内の名前の大文字/小文字を次のように設定します。
 - ・ 大文字 (VHDL)
 - デフォルトは小文字です。
 - ISE® Design Suite では大文字/小文字を次から設定します。
[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Case]
 - ・ maintain (Verilog)
 - デフォルトは [Maintain] です。
 - ISE Design Suite では大文字/小文字を次から設定します。
[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Case]
2. デザイン階層を保持します。

ISE Design Suite で[階層保持 \(KEEP_HIERARCHY\)](#)を設定します。
[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Keep Hierarchy]
3. 等価フリップフロップを保持します。
 - ・ XST では、等価フリップフロップがデフォルトで削除されます。
 - ・ ISE Design Suite で [等価レジスタの削除 \(EQUIVALENT_REGISTER_REMOVAL\)](#) を設定します。
[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Equivalent Register Removal]

ロジックとフリップフロップの複製の回避

フリップフロップのセット/リセット信号のファンアウトが多いとフリップフロップが複製されますが、この複製が行われないようにするには、次いずれかの操作を行います。

- ・ ISE® Design Suite で次を変更して、デザイン全体の最大ファンアウト値を大きい値に設定します。
[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Max Fanout]
または
- ・ **最大ファンアウト (MAX_FANOUT)** 属性を使用して、PCI コアの RST ポートに接続されている初期化信号に高いファンアウト値を設定します。次に例を示します。

```
max_fanout=2048
```

コアの自動読み込み機能のオフ

デフォルトでは、タイミングおよびエリア予測のためコアが自動的に読み込まれます。

- ・ PCI コアが読み込まれる場合は、次のようなロジックに対して最適化が実行されます。
 - タイミング要件を満たさないロジック
 - マップ中にエラーになるようなロジック
- ・ ISE® Design Suite で**コアの読み込み (READ_CORES)** プロパティをオフにして、PCI コアを読み込まないようにします。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Read Cores]

デザイン制約

XST デザイン制約を使用すると、次が可能になります。

- ・ デザイン目標を満たす
- ・ 最適な回路インプリメンテーションを達成
- ・ 次を制御可能
 - － 合成
 - － 配置
 - － 配線

デフォルトの合成アルゴリズムを調整し、経験を生かすことで、デザインに最適な結果が得られるようになっています。最初の合成でデザイン目標が満たされなかった場合は、XST デザイン制約を使用して別の方法で合成し直します。

制約の指定

制約を設定するには、次のような方法があります。

- ・ ISE® Design Suite
- ・ コマンド ライン
- ・ VHDL 属性
- ・ Verilog の属性とメタ コメント
- ・ XST Constraint File (XCF)

すべての属性がどのツールや方法でも指定できるわけではありません。特定の制約に対してツールや方法がリストされていない場合は、その方法では設定できないことを示しています。

ISE Design Suite およびコマンド ライン

合成のほとんどの側面をグローバルに制御するには、次で制約を指定します。

- ・ ISE Design Suite
- ・ [コマンド ライン モード](#)の run コマンド

VHDL 属性

- ・ VHDL で制約を指定するには、次を実行します。
 1. VHDL 属性を HDL ソースコードに直接含めます。
 2. VHDL 属性を個々のデザイン エLEMENT に適用します。
- ・ VHDL 属性を使用すると、次が制御できます。
 - 合成
 - 配置
 - 配線
- ・ VHDL 属性を使用してインスタンスシート済みデバイス プリミティブのプロパティを定義することはできません。VHDL ジェネリックまたは Verilog パラメーターを使用してください。

Verilog の属性とメタ コメント

Verilog で制約を指定するには、Verilog 属性をとメタ コメントを HDL ソースコードに挿入します。

- ・ Verilog 属性は、Verilog のメタ コメントよりも推奨されます。
- ・ Verilog 属性またはメタ コメントを使用してインスタンスシート済みデバイス プリミティブのプロパティを定義することはできません。VHDL ジェネリックまたは Verilog パラメーターを使用してください。

XST Constraint File (XCF)

制約は XCF (ザイリンクス制約ファイル) で指定できます。

XCF を使用してインスタンスシート済みデバイス プリミティブのプロパティを定義することはできません。VHDL ジェネリックまたは Verilog パラメーターを使用してください。

詳細情報

詳細は、次を参照してください。

- ・ [プリミティブ プロパティの指定](#)
インスタンスシート済みデバイス プリミティブのプロパティを指定する方法
- ・ [制約の優先順序](#)
同じ制約のインスタンス複数が次のいずれかの場合の適用される制約の決定方法
 - 別の入力方法を使用して設定
 - 異なるオブジェクトに設定

制約の優先順序

XST では、競合する制約が次のどちらかに設定されるかによって、制約の優先順序が変わります。

- ・ 異なる入力方法を使用して同じオブジェクトに設定
- ・ 異なるオブジェクトに設定

別の入力方法を使用して制約を設定

異なる入力方法を使用して同じオブジェクトに設定する場合は、優先順位は次のようになります。優先順位の高い方からリストします。

1. XCF (ザイリンクス制約ファイル)
2. HDL 属性
3. ISE® Design Suite の [Process Properties] ダイアログ ボックスまたはコマンド ライン

異なるオブジェクトに制約を設定

制約が異なるオブジェクトに設定される場合、優先順位は次のようになります。

- ・ ローカルに設定した制約の方がグローバルに設定された制約よりも優先
- ・ 信号またはインスタンの制約セットがそれを含むデザイン ユニットの同じ制約セットよりも優先

合成オプションの設定

XST の合成オプションは、次のように設定します。

- ・ [ISE Design Suite での合成オプションの設定](#)
- ・ [その他のコマンド ライン合成オプションの設定](#)
- ・ [デフォルト以外のデザイン目標およびストラテジに対する合成オプションの設定](#)

ISE Design Suite での合成オプションの設定

- ・ ISE® Design Suite から XST のオプションを設定するには
 1. [Design] ウィンドウの [Hierarchy] パネルから HDL ソース ファイルを選択します。
 - a. [Processes] ペインで [Synthesize - XST] を右クリックします。
 - b. [Process Properties] をクリックします。
 - c. 次のカテゴリのいずれかを選択します。
 - Synthesis Options
 - HDL Options
 - Xilinx® Specific Options
 2. [Process Properties] ダイアログ ボックスで [Property display level] を次のいずれかに設定します。
 - a. Standard
最もよく使用されるオプション
 - b. Advanced
すべての使用可能なオプション
 3. [Display switch names] をオンにし、各オプションに対応するコマンド ライン オプションも表示します。
- ・ XST のデフォルト オプションに戻すには、[Default] をクリックします。

その他のコマンド ライン合成オプションの設定

[Process Properties] ダイアログ ボックスにリストされるデフォルトのオプション以外にも、リストされていない XST コマンド ライン オプションを指定することができます。

1. [Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスを表示して、[Synthesis Options] をクリックします。
2. [Other XST Command Line Options] の [Value] 列に必要なコマンド ライン オプションを追加します。
3. オプションを複数指定する場合は、スペースで区切ります。
4. 構文は、「[XST のコマンド](#)」を参照してください。

デフォルト以外のデザイン目標およびストラテジに対する合成オプションの設定

ISE® Design Suite には、特定のオプションで XST を実行できる定義済みの目標やストラテジがあらかじめ含まれています。

- ・ これらの設定を使用すると、特定の最適化目標に合わせたオプションがオンになります。
- ・ この方法を使用すると、XST 制約すべての詳細を確認することなく、デフォルト以外の制約を設定を試すことができます。
- ・ デザイン目標やストラテジを作成したり、保存しておくには、[Project] → [Design Goals & Strategies] をクリックします。

VHDL の属性

VHDL 属性を使用すると、HDL ソース コードに直接制約を記述できます。

- ・ 属性 type では、属性値の種類を定義します。
- ・ XST で使用できるタイプは string のみです。
- ・ 属性は、エンティティまたはアーキテクチャで宣言します。
- ・ アーキテクチャで宣言した場合は、その属性はエンティティ宣言では使用できません。
- ・ オブジェクトリストは、識別子をカンマで区切ったリストです。
- ・ 使用できるオブジェクトタイプは次のとおりです。
 - エンティティ
 - アーキテクチャ
 - コンポーネント
 - ラベル
 - 信号
 - 変数
 - タイプ
- ・ 制約は VHDL のエンティティに適用すると、コンポーネント宣言にも適用することができます。

VHDL 属性の宣言例

```
attribute AttributeName : Type ;
```


VHDL 属性の指定例

```
attribute AttributeName of ObjectList : ObjectType is  
AttributeValue ;
```

VHDL 属性の構文例

```
attribute RLOC : string;
```

VHDL 属性の例

```
attribute RLOC of ul23 : label is "R11C1.S0" ;  
attribute bufg of my_signal : signal is "sr";
```

Verilog-2001 の属性

XST では Verilog-2001 属性文がサポートされています。

- ・ Verilog-2001 の属性文には、次の特徴があります。
 - 合成ツールなどのアプリケーションに特定の情報を渡します。
 - 次の中であればどこでも、演算子または信号に指定できます。
 - ◆ モジュール宣言
 - ◆ インスタンス化文
- ・ コンパイラでその他の属性宣言がサポートされていても、XST では無視されます。
- ・ Verilog 属性は、次の場合に使用できます。
 - 次のような個々のオブジェクトに制約を設定する場合
 - ◆ モジュール
 - ◆ インスタンス
 - ◆ ネット
 - 次の合成制約を設定する場合
 - ◆ フル ケース
 - ◆ パラレル ケース

Verilog-2001 の構文

```
(* attribute_name = attribute_value *)
```

- ・ 属性は * で囲みます。
- ・ attribute_value には、文字列を指定する必要があります。整数値やスカラ値は使用できません。
- ・ attribute_value は、二重引用符 (" ") で囲む必要があります。
- ・ デフォルトの値は 1 です。(* attribute_name *) は (* attribute_name = "1" *) と同じです。

属性の配置

属性は、次のいずれかの方法を使用して配置できます。

1. 次を信号、モジュール、またはインスタンス宣言の直前に入力します。
 - ・ 属性は宣言と同じ行で指定すると次のようになります。

```
(* ram_extract = "yes" *) reg [WIDTH-1:0] myRAM [SIZE-1:0];
```

- ・ 属性は宣言と別の行で指定すると次のようになります。

```
(* ram_extract = "yes" *) reg [WIDTH-1:0] myRAM [SIZE-1:0];
```

2. 同じ Verilog オブジェクトに適用された複数の属性のリストを指定します。
 - ・ 属性はカンマで区別します。

```
(* attribute_name1 = attribute_value1, attribute_name2 = attribute_value2 *)
```

- ・ 属性は () で囲みます。

```
(* attribute_name1 = attribute_value1 *) (*attribute_name2 = attribute_value2 *)
```

- ・ 読みやすくするため、次のように属性リストを複数行に分けることもできます。

```
(*      ram_extract = "yes", ram_style = "block" *)  
reg [WIDTH-1:0] myRAM [SIZE-1:0]
```

Verilog-2001 の制限

Verilog-2001 では、次に対する属性はサポートされていません。

- ・ 信号の宣言
- ・ ステートメント
- ・ ポートの接続
- ・ 論理演算子

Verilog のメタ コメント

- ・ メタ コメントを使用すると、制約を Verilog コードで指定できます。
- ・ ザイリンクスでは Verilog-2001 属性構文の使用をお勧めしています。
- ・ Verilog メタ コメント構文は次のようになります。

```
// synthesis attribute AttributeName [of] ObjectName [is]  
AttributeValue
```

- ・ 次の制約には、異なる構文を使用します。
 - フル ケース
 - パラレル ケース
 - Translate Off と Translate On
- ・ 詳細は、「Verilog-2001 の属性とメタ コメント」を参照してください。

Verilog メタ コメントのコード例

```
// synthesis attribute RLOC of u123 is R11C1.S0  
// synthesis attribute HU_SET u1 MY_SET  
// synthesis attribute bufg of my_clock is "clk"
```

XCF（ザイリンクス制約ファイル）

HDL ソース コードで XST 制約を指定する以外にも、XCF で指定する方法があります。

- ・ ISE® Design Suite
- ・ コマンド ライン

ISE Design Suite での XCF の指定

ISE Design Suite で XCF ファイルを指定するには、次の手順に従ってください。

1. [Design] ウィンドウの [Hierarchy] パネルから HDL ソース ファイルを選択します。
2. [Synthesize - XST] プロセスを右クリックします。
3. [Process Properties] をクリックします。
4. [Synthesis Options] をクリックします。
5. [Synthesis Constraints File] を編集します。
6. [Synthesis Constraints File] を確認します。

コマンドラインでの XCF の指定

- ・ コマンドラインで XCF を指定するには、run コマンドで合成制約ファイル (-uc) オプションを使用します。
- ・ コマンドラインからの XST の起動方法と run コマンドについては、「[XST コマンド](#)」を参照してください。

XCF 構文

- ・ XCF 構文を使用すると、制約を次のいずれかに指定できます。
 - － デザイン全体
 - － 特定のエンティティまたはモジュール
- ・ XCF 構文は UCF 構文を拡張したものです。制約はネットやインスタンスに同じように適用します。

制約の定義と適用

- ・ XCF 構文を使用すると、デザイン階層の特定レベルに制約を適用することができます。
 - － キーワード MODEL を使用して、制約を適用するエンティティ/モジュールを定義します。
 - － エンティティ/モジュールに制約を設定した場合、制約はそのエンティティ/モジュールの各インスタンスに適用されます。
- ・ 次のように定義します。
 - － ISE Design Suite
のプロセス プロパティ
 - － コマンド ラインからの run コマンド
- ・ 例外を XCF ファイルで指定します。XCF ファイルの制約は、指定されたモジュールにのみ適用され、その下位にあるサブモジュールには適用されません。
- ・ エンティティ/モジュール全体に制約を適用するには、次の構文を使用します。

```
MODEL entityname constraintname = constraintvalue;
```

INIT と NET の使用

エンティティ/モジュールの特定インスタンスまたは信号に制約を適用するには、キーワード INST または NET を使用します。XST では、VHDL 変数に適用される制約はサポートされません。

構文

```
BEGIN MODEL entityname
```

```
INST instancename constraintname = constraintvalue ;
```

```
NET signalname constraintname = constraintvalue ;
```

構文例

```
BEGIN MODEL crc32
  INST stopwatch opt_mode = area ;
  INST U2 ram_style = block ;
  NET myclock clock_buffer = true ;
  NET data_in iob = true ;
END;
```

ネイティブ UCF 制約とそれ以外の UCF 構文

XST でサポートされる制約には、次が含まれます。

- ・ [ネイティブ UCF 制約](#)
- ・ [ネイティブ以外の UCF 制約](#)

ネイティブ UCF 制約

ネイティブ UCF の構文が使用されるのは、タイミング制約とエリア グループ制約のみです。

- ・ UCF 構文には、ワイルドカードおよび階層名が含まれます。
- ・ 次のようなネイティブ UCF 制約には、UCF 構文を使用します。
 - 周期 (PERIOD)
 - オフセット (OFFSET)
 - FROM-TO
 - タイミング名 (TNM)
 - ネットのタイミング名 (TNM_NET)
 - タイムグループ (TIMEGROUP)
 - タイミング無視 (TIG)
- ・ これらの制約を BEGIN MODEL と END 間で使用すると、エラー メッセージが表示されます。

ネイティブ以外の UCF 制約

次のようなネイティブ以外の UCF 制約の場合は、MODEL または BEGIN MODEL... END; 構文を使用します。

含まれる制約

ネイティブ以外の UCF 制約には、次が含まれます。

- ・ 次のような純粋な XST 制約
 - FSM 自動抽出
 - RAM スタイル
- ・ タイミング以外のインプリメンテーション制約
 - RLOC
 - キープ

デフォルトの階層区切り文字

XST では、デフォルトの階層区切り文字はスラッシュ (/) です。

- ・ XST Constraint File (XCF) で階層インスタンスやネット名にタイミング制約を指定する際は、このデフォルトの階層区切り文字を使用します。
- ・ 階層区切り文字を変更するには、[Hierarchy Separator] (-hierarchy_separator) オプションを使用します。

XCF 構文の制限

- ・ XCF (ザイリンクス制約ファイル) の構文では、次がサポートされません。
 - MODEL 文のネスト
 - タイミング制約以外の制約でのインスタンス名や信号名のワイルドカード
 - 一部のネイティブ UCF 制約
 - 階層インスタンス名または信号名
- ・ BEGIN MODEL と END 間にリストされたインスタンス名や信号名のみが、エンティティ内で有効です。
- ・ 詳細は、『[制約ガイド](#)』(UG625) を参照してください。

XCF で指定されたタイミング制約

- ・ 次の XST タイミング制約は、XCF (ザイリンクス制約ファイル) からのみ設定可能です。
 - [周期 \(PERIOD\)](#)
 - [オフセット \(OFFSET\)](#)
 - [FROM-TO](#)
 - [タイミング名 \(TNM\)](#)
 - [ネットのタイミング名 \(TNM_NET\)](#)
 - [タイムグループ \(TIMEGROUP\)](#)
 - [タイミング無視 \(TIG\)](#)
 - [タイミング仕様 \(TIMESPEC\)](#)
- ・ 『[制約ガイド](#)』(UG625) を参照してください。
- ・ タイミング仕様識別子 (TSidentifier)
 - ・ 『[制約ガイド](#)』(UG625) を参照してください。
- ・ これらのタイミング制約には、次のような特徴があります。
 - インプリメンテーション ツールに伝搬されない
 - XST で認識されます。
 - 合成最適化に影響します。
- ・ PAR (配置配線) にこれらの制約を渡すには、[タイミング制約の書き込み \(-write_timing_constraints\)](#) を使用します。
- ・ 各制約の値とターゲットについての詳細は、『[制約ガイド](#)』を参照してください。

一般制約

この章では、XST の一般制約について説明しています。

この章には、ほとんどの制約の次の情報を含めます。

- ・ 制約の説明
- ・ 適用可能エレメント
- ・ 適用ルール
- ・ 制約値
- ・ 構文例

I/O バッファの追加

I/O バッファの追加 (**-iobuf**) コマンドライン オプションを使用すると、I/O バッファの挿入を有効または無効にできます。

- ・ 後でインスタンスエートするデザインの一部を合成できます。
- ・ XST では、I/O バッファがデザインに自動的に挿入されます。
- ・ I/O バッファを一部の I/O に手動でインスタンスエートする場合、XST では残りの I/O にのみ I/O バッファが挿入されます。
- ・ デザインに I/O バッファを追加した場合は、デザインを別のデザインのサブモジュールとして使用することはできません。
- ・ I/O バッファが XST で挿入されないようにするには、**-iobuf** オプションを **no** に設定します。

適用可能エレメント

グローバルに適用します。

適用ルール

デザインのプライマリ I/O に適用されます。

制約値

- ・ **yes** (デフォルト)
yes を選択すると、IBUF および OBUF プリミティブが生成されます。IBUF/OBUF プリミティブは、最上位モジュールの I/O ポートに接続されます。
- ・ **no**
大型デザインの後でインスタンスエートされる内部モジュールを合成する場合は、このオプションを **no** に設定する必要があります。
- ・ **true**
- ・ **false**
- ・ **soft**

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンドライン構文例

run コマンドでグローバルに設定します。

```
-iobuf {yes|no|true|false|soft}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Xilinx-Specific Options] → [Add I/O Buffers]

ボックス タイプ

ボックス タイプ (BOX_TYPE) 制約を使用すると、モジュールのビヘイビアを合成しないようにできます。

- ・ ボックス タイプには、次のような特徴があります。
 - 合成制約です。
 - コンポーネントに適用できます。
- ・ 少なくともブロックの 1 つのインスタンスに BOX_TYPE 制約を設定すると、デザインすべてのインスタンスに制約が適用されます。

適用可能エレメント

次のデザイン エレメントに適用されます。

- ・ VHDL
component、entity
- ・ Verilog
module、instance
- ・ XCF
model、instance

適用ルール

設定したデザイン エレメントに適用されます。

制約値

- ・ **primitive**
ログ ファイルにブラック ボックスの推論がレポートされません。
- ・ **black_box**
primitive と同じです。今後使用できなくなる予定なのでご注意ください。
- ・ **user_black_box**
ログ ファイルにブラック ボックスの推論がレポートされます。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute box_type: string;
```

次のように指定します。

```
attribute box_type of  
  {component_name|entity_name}:{component|entity} is  
  "{primitive|black_box|user_black_box}";
```

Verilog の構文例

次をインスタンスエーションの直前に入力します。

```
(* box_type = "{primitive|black_box|user_black_box}" *)
```

XCF の構文例 1

```
MODEL "entity_name"  
box_type="{primitive|black_box|user_black_box}";
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
INST " instance_name"  
box_type="{primitive|black_box|user_black_box}"; END;
```

バス区切り文字

バス区切り文字 (**-bus_delimiter**) コマンドライン オプションを使用すると、信号ベクターをネットリストに書き込む際に使用する形式を指定できます。

適用可能エレメント

構文に適用されます。

適用ルール

ありません。

制約値

- ・ <> (デフォルト)
- ・ []
- ・ {}
- ・ ()

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-bus_delimiter {<>|[]|{}|() }
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Bus Delimiter]

大文字/小文字の区別

大文字/小文字の区別 (**-case**) コマンド ライン オプションでは、次を指定できます。

- ・ インスタンス名およびネット名を最終的なネットリストに記述する際に、名前にすべて大文字を使用するか、小文字を使用するかを指定します。

または

- ・ ソースの文字を保持するかどうかを指定します。

ソースの文字は、Verilog または VHDL 合成フローで保持できます。

適用可能エレメント

構文に適用されます。

適用ルール

ありません。

制約値

- ・ upper
- ・ lower
- ・ maintain (デフォルト)

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-case {upper|lower|maintain}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Case]

Case 文のインプリメンテーション形式

Case 文のインプリメンテーション形式 (-vlgcase) コマンドライン オプションには、次の特徴があります。

- ・ Verilog デザインのみをサポートします。
- ・ XST で Verilog の case 文をどのように解釈させるかが指定できます。

適用可能エレメント

グローバルに適用します。

適用ルール

ありません。

制約値

- ・ full
 - XST では、case 文が完了していると認識されます。
 - ラッチは作成されません。
- ・ parallel
 - XST では分岐はパラレルに発生できないと判断されます。
 - プライオリティ エンコーダーは使用されません。
- ・ full-parallel
 - XST では、case 文が完了していると認識されます。
 - XST では分岐はパラレルに発生できないと判断されます。
 - ラッチとプライオリティ エンコーダーが節約されます。
- ・ None
 - case 文のビヘイビアーを記述どおりにインプリメントします。デフォルトの値はありません。

詳細は、次を参照してください。

- ・ [フル ケース](#)
- ・ [パラレル ケース](#)
- ・ [マルチプレクサ](#)

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-vlgcase {full|parallel|full-parallel}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [Case Implementation Style]

複製接尾語の設定

複製接尾語の設定 (-duplication_suffix) には、次の特徴があります。

- ・ フリップフロップが複製されたときの名前の付け方を設定できます。
- ・ デフォルト名の終わりにテキスト スtringを追加できます。

複製されたフリップフロップの命名

XST でフリップフロップが複製される際、新しいフリップフロップに名前が付けられます。

- ・ 元のフリップフロップ名の終わりに `_n` が付きます。
- ・ この `n` は整数になります。

複製されたフリップフロップの命名例

- ・ 元のフリップフロップ名は `my_ff` です。
- ・ フリップフロップは 3 回複製されます。
- ・ XST では、次の名前がフリップフロップが生成されます。
 - `my_ff_1`
 - `my_ff_2`
 - `my_ff_3`

テキスト スtringの指定

この制約を使用すると、デフォルト名の終わりにテキスト スtringを追加できます。

テキスト スtringの指定例 1

- ・ インデックス番号が表示される部分は、エスケープ文字 `%d` を使用します。
- ・ フリップフロップの名前が `my_ff` の場合に `_dupreg%d` と指定すると、XST は次のように名前を付けます。
 - `my_ff_dupreg_1`
 - `my_ff_dupreg_2`
 - `my_ff_dupreg_3`

テキスト スtringの指定例 2

- ・ エスケープ文字 `%d` は接尾語の定義のどこにでも挿入できます。
- ・ `_dup%d_reg` に指定した場合は、XST では次の名前が付けられます。
 - `my_ff_dup_1_reg`
 - `my_ff_dup_2_reg`
 - `my_ff_dup_3_reg`

適用可能エレメント

ファイルに適用されます。

適用ルール

ありません。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-duplication_suffix string%dstring
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Other]

フル ケース

フル ケース (FULL_CASE) 制約には、次の特徴があります。

- ・ Verilog デザインのみに適用されます。
- ・ case、casex または casez 文で、セレクトのすべての値が記述されていることを示します。
- ・ 記述されていない条件がある場合にハードウェアが作成されません。

詳細は「[マルチプレクサ](#)」を参照してください。

適用可能エレメント

Verilog メタ コメントの case 文に適用されます。

適用ルール

ありません。

制約値

- ・ **full**
- ・ **parallel**
- ・ **full-parallel**

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。
詳細は、「[はじめに](#)」の構文例を参照してください。

Verilog の構文例

(* full_case *)

- ・ コマンドにはターゲット参照が含まれないため、セレクトのすぐ後ろに属性を記述します。

```
(* full_case *)
casex select
  4'b1xxx: res = data1;
  4'b01xx: res = data2;
  4'b0xx1x: res = data3;
  4'b0xxx1: res = data4;
endcase
```

- ・ これは Verilog コードのメタコメントとしても使用可能です。メタコメントの構文は、次のように通常のメタコメントと異なります。

```
// synthesis full_case
```

- ・ コマンドにはターゲット参照が含まれないため、セレクトのすぐ後ろにメタコメントを記述します。

```
casex select // synthesis full_case
  4'b1xxx: res = data1;
  4'b01xx: res = data2;
  4'b0xx1x: res = data3;
  4'b0xxx1: res = data4;
endcase
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

-vlgcase {full|parallel|full-parallel}

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Full Case] をクリックします。

[Case Implementation Style] には [Full] を選択します。

RTL 回路図の生成

RTL 回路図の生成 (-rtlview) コマンドライン オプションを使用すると、ネットリスト ファイルを生成できます。

- ・ このネットリスト ファイルは、デザインの Register Transfer Level (RTL) 構造を表します。
- ・ ネットリスト ファイルは、次から確認します。
 - RTL Viewer
 - Technology Viewer
- ・ RTL 表示を含むファイルの拡張子は .ngr です。

適用可能エレメント

ファイルに適用されます。

適用ルール

ありません。

制約値

- ・ yes
 - ・ no
 - ・ only
- only に設定すると、合成プロセスは RTL 表示が生成された直後に停止します。

デフォルト設定

デフォルトは、入力方法によって異なります。次を参照してください。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンドライン構文例

run コマンドでグローバルに設定します。

```
-rtlview {yes|no|only}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Generate RTL Schematic]

デフォルトは、yes です。

ジェネリック

ジェネリック (-generics) コマンド ライン オプションを使用すると、最上位レベルのブロックで次の値を再定義できます。

- ・ ジェネリック (VHDL)
- ・ パラメーター (Verilog)

ジェネリック オプションを使用してこれらの値を再定義すると、次が可能になります。

- ・ ソースコードを変更しなくてもデザインを簡単に修正できます。
- ・ IP コアの生成やフロー テストで使用できます。

値の不一致

再定義された値と HDL ソースコードで定義された値が異なると、XST では次のように処理されます。

- ・ 再定義された値がデザインに存在しないジェネリックまたはパラメーター名を使用する場合：
 - － 警告メッセージは表示されません。
 - － コマンドラインの定義が無視されます。
- ・ 再定義された値が HDL ソースコードのデータ型と一致しない場合：
 - － 警告メッセージが表示されます。
 - － コマンドラインの定義が無視されます。
- ・ XST で不一致が検出できなかった場合：
 - － 警告メッセージは表示されません。
 - － XST は再定義された値を HDL ソースコードのデータ型に適用しようとします。

適用可能エレメント

グローバルに適用します。

適用ルール

ありません。

制約値

- ・ name
最上位レベルのデザイン ブロックのジェネリックまたはパラメーターの名前
- ・ value
最上位レベルのデザイン ブロックのジェネリックまたはパラメーターの値

デフォルト設定

デフォルトでは何も定義されていません。

`-generics {}`

制約の構文ガイドライン

- ・ 2 進数、16 進数、10 進数の場合、接頭語とその値の間にはスペースは入りません。

```
-generics {company="mycompany" width=5 init_vector=b100101}
```

- ・ このコマンドは、次を設定しています。

- company を mycompany に設定
- width を 5 に設定
- init_vector を b100101 に設定

- ・ 次のタイプのジェネリックの値を 2 進数または 16 進数で指定します。

- **std_logic_vector**
- **std_ulogic_vector**
- **bit_vector**

例

必須の基本接頭辞 (b) なしで 2 進数値を指定すると、XST ではジェネリック タイプが integer であると仮定されます。XST では、次のような不一致を示すエラー メッセージが表示されます。

```
ERROR:HDLCompiler:839 - "example.vhd" Line 11: Type
std_logic_vector does not match with the integer literal
```

- ・ フォーマットは、ジェネリックの値の型によって次の表に示すように異なります。
 - ・ {} 内に名前/値を挿入します。
 - ・ 名前/値のペアはそれぞれスペースで区切ります。
 - ・ XST にはスカラ型の定数しか値として使用できません。複合データ型 (アレイまたはレコード) は次の場合にしか使用できません。
- string
 - std_logic_vector
 - std_ulogic_vector
 - signed
 - unsigned
 - bit_vector

ジェネリック値の構文例

タイプ	ジェネリック値の構文例
文字列	"mystring"
2 進数	b00111010
16 進数	h3A
10 進数 (整数)	d58 (または 58)
論理値 - 真	TRUE
論理値 - 偽	FALSE

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。
詳細は、「はじめに」の構文例を参照してください。

XST コマンド ライン構文例

```
run -generics {name=value name=value ...}
```


HDL ライブラリ マップ ファイル

HDL ライブラリ マップ ファイル (-xsthdpini) を使用すると、ライブラリ マップを定義できます。

- ・ ライブラリ マップ ファイルには、次が含まれます。
 - 次の 2 つのパラメーター
 - ◆ XSTHDPINI
 - ◆ XSTHDPDIR
 - 内容：
 - ◆ ライブラリ名
 - ◆ ライブラリがコンパイルされたディレクトリ名
- ・ XST では、次の 2 つのライブラリ マップ ファイルが維持されます。
 - インストール済みの INI ファイル
 - カスタム INI ファイル

インストール済みの INI ファイル

```
-- Default lib mapping for XST
std=$XILINX/vhdl/xst/std
ieee=$XILINX/vhdl/xst/unisim
unisim=$XILINX/vhdl/xst/unisim
aim=$XILINX/vhdl/xst/aim
pls=$XILINX/vhdl/xst/pls
```

- ・ インストール済みの INI ファイル：
 - ファイル名は xhdp.ini です。
 - ザイリンクスのソフトウェア インストール中にインストールされます。
 - デフォルトです。
 - %XILINX%\vhdl\xst に含まれます。
 - 標準 VHDL および UNISIM ライブラリの場所に関する情報が含まれます。
 - 修正できません。
- 注記：** 構文はユーザー ライブラリ マップに使用できます。
- 次のように表示されます。
- ・ この INI ファイルのフォーマットを使用して、独自のライブラリの場所を定義できます。デフォルトでは、コンパイルされた VHDL ファイルはプロジェクト ディレクトリの xst サブディレクトリに保存されます。

カスタム INI ファイル

- ・ ユーザー プロジェクト専用カスタムの INI ファイル定義できます。
- ・ カスタム INI ファイルの場所は、次のいずれかの方法で指定できます。
 - ISE® Design Suite から VHDL の INI ファイルを選択します。
[Process Properties] ダイアログ ボックス → [Synthesis Options]、または
 - 次のコマンドをスタンドアロン モードで使用して、コマンドラインから **-xsthdpini** オプションを指定します。

set -xsthdpini file_name
- ・ このライブラリ マップ ファイルの名前は任意ですが、拡張子は .ini にすることをお勧めします。ファイルのフォーマットは次のとおりです。
 - *library_name=path_to_compiled_directory*
 - コメント行にはハイフン 2 つ (--) を使用します。

MY.INI ファイル例

```
work1=H:\Users\conf\my_lib\work1
work2=C:\mylib\work2
```

適用可能エレメント

ファイルに適用されます。

適用ルール

ありません。

制約値

指定できる値はディレクトリ名です。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
set -xsthdpini file_name
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [VHDL INI File]

この制約を表示するには、[Edit] → [Preferences] → [Processes] をクリックし、[Property display level] を [Advanced] に設定します。

階層区切り文字

階層区切り文字の指定 (-hierarchy_separator) コマンドライン オプションを使用すると、デザイン階層をフラット化した際の名前の生成で使用する階層の区切り文字を指定できます。

- ・ 次の文字がサポートされます。
 - _ (アンダースコア)
 - / (スラッシュ)
- ・ / (スラッシュ) を使用すると、次のような利点があります。
 - デザインのデバッグに便利
 - 階層の場合名前を識別しやすくなる

階層区切り文字の例

- ・ デザインにインスタンス INST1 というサブブロックがあります。
- ・ このサブブロックに TMP_NET というネットが含まれているとします。
- ・ 階層はフラット化されます。
- ・ TMP_NET は INST1_TMP_NET という名前になります。
- ・ 階層区切り文字はスラッシュ (/) です。
- ・ ネット名は NST1/TMP_NET です。

適用可能エレメント

ファイルに適用されます。

適用ルール

ありません。

制約値

- ・ _ (アンダースコア)
- ・ / (スラッシュ)

デフォルトの制約値

新規プロジェクトでのデフォルトは / (スラッシュ) です。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-hierarchy_separator {_|/}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Hierarchy Separator]

合成制約ファイルの無視

合成制約ファイルの無視 (-iuc) コマンドライン オプションを使用すると、[合成制約ファイル \(-uc\)](#) で指定した制約ファイルが無視されます。

適用可能エレメント

ファイルに適用されます。

適用ルール

ありません。

制約値

- ・ yes
- ・ no

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、[「はじめに」](#)の構文例を参照してください。

XST コマンドライン構文例

run コマンドでグローバルに設定します。

```
-iuc {yes|no}
```

ISE Design Suite での設定

注意：この制約は、ISE® Design Suite では [Synthesis Constraints File] と表示されます。制約ファイルはこのオプションをオフにすると無視されます。デフォルトではオンになっており (コマンドライン オプションの -iuc no と同じ)、指定した合成制約ファイルが考慮されます。

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Synthesis Constraints File]

I/O 規格

I/O 規格 (IOSTANDARD) 制約は、I/O プリミティブに I/O 規格を割り当てるために使用します。

- ・ 次を使用して信号やインスタンスに個別に適用できます。
 - VHDL 属性
 - Verilog 属性
 - XCF 制約
- ・ この制約はグローバルには適用できません。

この制約の詳細は、『[制約ガイド](#)』(UG625)を参照してください。

キープ

ネットリストで信号を保持するために使用します。

KEEP 制約には、次の特徴があります。

- ・ 高度なマップ制約です。
- ・ 次を使用した信号に適用されます。
 - VHDL 属性
 - Verilog 属性
 - XCF 制約

この制約の詳細は、『[制約ガイド](#)』(UG625)を参照してください。

ネットの吸収

- ・ デザインのマップ時に、一部のネットが論理ブロックに含まれることがあります。
- ・ ブロックに含まれたネットは、物理的なデザイン データベースには存在しなくなります。次に例を示します。
 - ネットの各サイドに接続されたコンポーネントが同じ論理ブロックにマップされる場合などに発生することがあります。
 - この後、ネットはそのコンポーネントを含むブロックに含まれます。

KEEP の制限

- ・ KEEP 制約を使用すると、最終ネットリストに信号は保持されますが、周囲のロジックが保持されるわけではありません。
 - 周囲のロジックは、XST の最適化により変換されている可能性があります。
 - 下記の「KEEP の制限例」を参照してください。
- ・ 信号とその周囲のエLEMENTの両方を保持するには、[保存 \(S または SAVE\)](#) を使用する必要があります。
- ・ KEEP は、レジスタの複製を制御するために使用しないでください。この場合、[レジスタの複製制約](#)を使用してください。
- ・ KEEP は、等価レジスタの削除を制御するために使用しないでください。この場合は、[等価レジスタの削除制約](#)を使用してください。

KEEP の制限例

- ・ KEEP を 4:1 マルチプレクサの 2 ビット セレクターに KEEP を付けると、最終のネットリストでその信号が保持されます。
- ・ このマルチプレクサが XST によりワンホット エンコードを使用して再エンコードされていると、最終ネットリストでは元の 2 ビットではなく 4 ビット幅になります。
- ・ 信号の構造を保持するには、KEEP 制約に加えて [列挙型エンコーディング \(ENUM_ENCODING\)](#) 制約も使用する必要があります。

制約値

- ・ true
 - 該当する信号が NGC ネットリストに保持されます。
 - KEEP がネットリストに渡されます。
 - インプリメンテーションでその信号が保持されます。
- ・ soft
 - 該当する信号が NGC ネットリストに保持されます。
 - KEEP はインプリメンテーションには渡されません。
 - 信号は最適化でなくなる可能性があります。

注記： XCF (ザイリンクス制約ファイル) では、キープ (KEEP) 制約の値はオプションで二重引用符 (") で囲むことができます。soft の場合は、必ず二重引用符を使用する必要があります。

- ・ false
 - 該当する信号は保持されません。
 - 内部信号保持規則の結果、信号は NGC ネットリストにまだ残ります。
 - これらの規則以外では何もされません。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XCF の構文例 1

```
BEGIN MODEL testkeep  
NET aux1 KEEP=true;  
END;
```

XCF の構文例 2

```
BEGIN MODEL testkeep  
NET aux1 KEEP="soft";  
END;
```


階層の維持

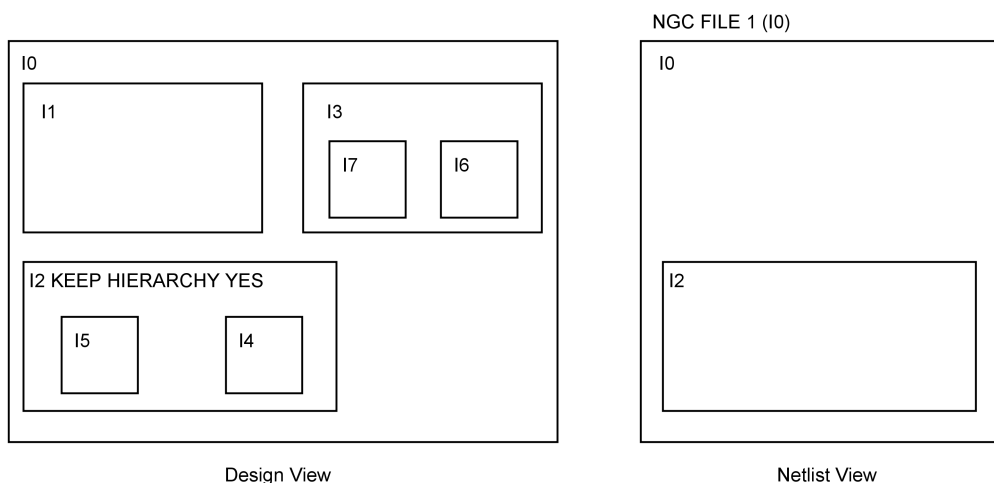
階層の維持 (KEEP_HIERARCHY) 制約は、HDL デザインで指定された階層ブロック (VHDL エンティティおよび Verilog モジュール) に関連します。

- ・ この制約には、次のような特徴があります。
 - 合成制約およびインプリメンテーション制約です。
 - HDL 合成ツールで推論されたマクロには適用されません。
- ・ デザイン階層が合成で保持された場合、インプリメンテーションでもこの制約を使用して次が実行されます。
 - インプリメンテーションで階層が保持されます。
 - 該当する階層を使用してシミュレーション ネットリストが作成されます。
- ・ XST では、最適な結果を得るために、エンティティおよびモジュールの境界を最適化してデザインをフラットにすることがあります。
- ・ yes に設定されると、生成されるネットリストは次のようになります。
 - 階層になります。
 - すべてのエンティティおよびモジュールの階層およびインターフェイスに対応します。

階層の保持

- ・ HDL デザインは階層ブロックの集合です。より単純な階層で個別に最適化が行われるため、デザイン階層を保持すると処理速度が向上します。
- ・ 次が可能になるので、階層ブロックのマージによりフィットの結果が向上します
 - 積項の少量化
 - デバイス マクロセルの少量化
 - 周波数の向上
- ・ このように改善されるのは、コラプスや因数分解などの最適化のプロセスがロジック全体にグローバルに適用されるためです。
- ・ エンティティまたはモジュール **I2** に KEEP_HIERARCHY 制約を設定した場合：
 - I2 の階層は最終ネットリストに含まれます。
 - I2 の中にある I4 および I5 はフラットになります。
 - また、I1、I3、I6、I7 も同様にフラット化されます。

階層維持の図



X9542

適用可能エレメント

階層ブロックまたはシンボル ブロックを含む論理ブロックに適用します。

適用ルール

設定したエンティティまたはモジュールに適用されます。

制約値

- ・ yes
 - － HDL プロジェクトで記述されたデザイン階層を保持します。
 - － yes を合成に適用した場合、インプリメンテーションにも適用されます。
- ・ no

階層ブロックが 最上位モジュールにマージされます。
- ・ soft
 - － 合成でデザインの階層を保持します。
 - － 制約はインプリメンテーションには渡されません。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute keep_hierarchy : string;
```

次のように指定します。

```
attribute keep_hierarchy of architecture_name: architecture is  
"{yes|no|soft}";
```

Verilog の構文例

```
(* keep_hierarchy = "{yes|no|soft}" *)
```

XCF の構文例

```
MODEL "entity_name" keep_hierarchy={yes|no|soft};
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-keep_hierarchy {yes|no|soft}
```

詳細は、「[コマンドライン モード](#)」を参照してください。

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Keep Hierarchy]

ライブラリ検索順

ライブラリ検索順 (**-lso**) コマンドライン オプションでは、ライブラリ ファイルを使用する順序を指定します。

これは、次のいずれかの方法で指定できます。

- ・ 次のいずれかの方法で ISE® Design Suite でライブラリ検索順 (LSO) ファイルを指定します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Library Search Order] を選択します。

- ・ **-lso** コマンドライン オプションを使用します。

詳細は、「[ライブラリ検索順 \(LSO\) ファイル](#)」を参照してください。

適用可能エレメント

ファイルに適用されます。

適用ルール

ありません。

制約値

使用できる値はファイル名のみです。

デフォルト設定

デフォルトのファイル名はありません。このオプションを指定しない場合は、デフォルトのライブラリ検索順が使用されます。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「[はじめに](#)」の構文例を参照してください。

XST コマンドライン構文例

run コマンドでグローバルに設定します。

```
-lso file_name.lso
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Library Search Order] を選択します。

LOC

LOC は、デバイス内のデザイン エLEMENTを配置する位置 (ロケーション) を定義します。
この制約の詳細は、『[制約ガイド](#)』(UG625)を参照してください。

ネットリスト階層

ネットリスト階層 (-netlist_hierarchy) コマンド ライン オプションには、次の特徴があります。

- ・ 最終の NGC ネットリスト ファイルが生成される形式を制御できます。
- ・ 最適化が一部のみ終了している場合や、デザインが完全にフラット化された場合にも、階層ネットリストを書き出すことができます。

階層は常に完全に再構築されるわけではありません。

LUT 最適化中の小さな階層ブロックの最適化

- ・ 下位レベル合成中、LUT パッキングなどのロジック最適化により、一部の小さな階層ブロックのすべてのロジックが周囲に送信されることがあります。
- ・ これらの小さな階層ブロックには、次のような特徴があります。
 - かなり単純
 - 通常 2 ～ 3 個の LUT
 - 最適化中に削除
 - 最終ネットリストで再構築なし

階層を越えたマクロのグループ化

- ・ アドバンス HDL 合成中、XST では基本的な推論済みマクロがより高度で複雑なマクロにまとめようとされます。
- ・ これらの統合されたマクロは、通常 DSP またはブロック RAM リソースを使用してインプリメンテーションされます。
- ・ グループ化されたマクロが階層ブロック内で推論されると、ローカルの階層バウンダリが次のようになります。
 - 削除されることがあり
 - 最終ネットリストで再構築なし

適用可能エレメント

グローバルに適用します。

適用ルール

ありません。

制約値

- ・ as_optimized (デフォルト)
 - XST では、[階層の維持 \(KEEP_HIERARCHY\)](#) が考慮されます。
 - NGC ネットリストを最適化された形式で生成します。
 - 階層ブロックにはフラット化できるものもあれば、階層バウンダリを維持したままフラット化できないものもあります。
- ・ rebuilt

XST では、[階層の維持 \(KEEP_HIERARCHY\)](#) 制約に関係なく、階層的な NGC ネットリストが書き出されます。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「[はじめに](#)」の構文例を参照してください。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
- netlist_hierarchy {as_optimized|rebuilt}
```

最適化エフォート

最適化エフォート (OPT_LEVEL) 制約では、合成最適化のエフォートレベルを指定します。

適用可能エレメント

グローバルに適用するか、エンティティまたはモジュールに適用します。

適用ルール

設定したエンティティまたはモジュールに適用されます。

制約値

- ・ 1 (標準最適化エフォートレベル) (デフォルト)
 - デフォルトの最適化エフォートレベルが推奨されます。
 - 特に階層デザインで、次のようになります。
 - ◆ ハイレベルの最適化
 - ◆ 処理時間が高速
- ・ 2 (高度最適化エフォートレベル)
 - 最適化手法を追加で検索するように XST に命令します。
 - 合成ランタイムがかなり増加してしまうことがあります。
 - 必ずしも結果が改善されるわけではありません。
 - 一部のデザインにのみ利点があります。デザインによっては、まったく改善がない場合や、結果が悪化することすらあります。
- ・ 0 (高速最適化エフォートレベル)
 - 最適化エフォートレベル 1 (標準) で使用される最適化アルゴリズムの一部をオフにします。
 - 合成を最短のランタイムで実行します。
 - 最適化のトレードオフが発生することもあります。
 - 結果を素早く取得するため、設計の初期段階で使用されます。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute opt_level: string;
```

次のように指定します。

```
attribute opt_level of entity_name: entity is "{0|1|2}";
```

Verilog の構文例

```
(* opt_level = "{0|1|2}" *)
```


XCF の構文例

```
MODEL "entity_name" opt_level={0|1|2};
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-opt_level {0|1|2}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Optimization Effort]

最適化目標

最適化目標 (OPT_MODE) 制約では、合成の最適化方法を指定します。

適用可能エレメント

グローバルに適用するか、エンティティまたはモジュールに適用します。

適用ルール

設定したエンティティまたはモジュールに適用されます。

制約値

- ・ speed (デフォルト)
ロジック レベル数を削減されるので、周波数が上がります。
- ・ area
デザイン インプリメンテーションで使用するロジック総数の低減されるため、デザイン
フィットが向上します。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。
詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute opt_mode: string;
```

次のように指定します。

```
attribute opt_mode of entity_name: entity is "{speed|area}";
```

Verilog の構文例

```
(* opt_mode = "{speed|area}" *)
```

XCF の構文例

```
MODEL "entity_name" opt_mode={speed|area};
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-opt_mode {area|speed}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Optimization Goal]

パラレル ケース

パラレル ケース (PARALLEL_CASE) 制約には、次の特徴があります。

- ・ Verilog デザインにのみ使用できます。
- ・ case 文がパラレル マルチプレクサとして合成されるよう指定します。
- ・ case 文が優先順位付きのカスケードされた if-elsif 文に変換されないようにします。

適用可能エレメント

Verilog メタ コメントの case 文にのみ適用されます。

適用ルール

ありません。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

Verilog の構文例

(* parallel_case *)

- ・ このコマンドにはターゲット参照が含まれないため、セレクトのすぐ後ろに属性を記述します。

```
(* parallel_case *)
casex select
  4'b1xxx: res = data1;
  4'b01xx: res = data2;
  4'b0xx1x: res = data3;
  4'b0xxx1: res = data4;
endcase
```

- ・ この制約は、Verilog でメタ コメントとしても使用できます。メタ コメントの構文は、次のように通常のメタ コメントと異なります。

// synthesis parallel_case

- ・ このコマンドにはターゲット参照が含まれないため、セレクトのすぐ後ろにメタ コメントを記述します。

```
casex select // synthesis parallel_case
  4'b1xxx: res = data1;
  4'b01xx: res = data2;
  4'b0xx1x: res = data3;
  4'b0xxx1: res = data4;
endcase
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

-vlgcase {full|parallel|full-parallel}

RLOC

RLOC 制約には、次の特徴があります。

- ・ 基本的なマップおよび配置制約です。
- ・ ロジック エlementを独立した集合にグループ化します。
- ・ Elementの位置は、デザイン全体の最終的な配置に関係なく、同じ集合内のほかのElementに相対的に定義できます。

この制約の詳細は、『[制約ガイド](#)』(UG625)を参照してください。

保存

保存 (S または SAVE) 制約は、高度なマップ制約です。

- ・ デザインがマップされると、次が実行されます。
 - － 一部のネットが論理ブロックに吸収される
 - － LUT のようなエレメントが最適化で削除される
- ・ SAVE 制約を使用すると、このような最適化が回避でき、合成後のネットリストで特定のネットおよびブロックへのアクセスを保持できます。
- ・ 次のような最適化手法をオフにします。
 - － ネットやブロックの複製
 - － レジスタ自動調整

SAVE の適用先	XST の動作
ネット	これらのエレメントに接続されたネットを含め、ネットとそのネットに直接接続されたエレメントすべてが最終ネットリストに保持されます。
ブロック	LUT とそれに接続された信号すべてが保持されます。

この制約の詳細は、『[制約ガイド](#)』(UG625)を参照してください。

適用可能エレメント

- ・ ネット

S (SAVE) 制約がネットに適用されると、そのネットは直接接続されたエレメントすべてと共に最終ネットリストに保存されます。これらのエレメントに接続されたネットも保存されます。
- ・ インスタンス化されたデバイス プリミティブ

SAVE 制約が LUT のようなインスタンス化されたプリミティブに適用されると、その LUT とそれに直接接続された信号すべてが保持されます。

合成制約ファイル

合成制約ファイルの指定 (-uc) コマンドライン オプションを使用すると、合成中に XST で使用されるザイリンクス制約ファイル (XCF) を指定できます。

- ・ 制約ファイルの拡張子は .xcf です。
- ・ 拡張子が異なるファイルを指定するとエラーが発生し、プロセスが中止されます。

適用可能エレメント

ファイルに適用されます。

適用ルール

ありません。

制約値

使用できる値はファイル名のみです。デフォルト値はありません。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-uc filename
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Synthesis Constraints File]

Translate Off と Translate On

TRANSLATE_OFF と TRANSLATE_ON を使用すると、シミュレーション コードなど、合成に関係のない HDL ソース コードを無視するよう指定できます。

- ・ TRANSLATE_OFF: 無視するセクションの冒頭を指定
- ・ TRANSLATE_ON: 無視するセクションの終わりを指定

Synopsys 制約

Translate Off および Translate On は、Synopsys 制約です。

- ・ XST では Verilog で Translate Off および Translate On 制約がサポートされます。
- ・ 自動変換も VHDL および Verilog の両方で使用できます。
- ・ TRANSLATE_OFF および TRANSLATE_ON は、次のワードと共に使用できます。
 - **synthesis**
 - **synopsys**
 - **pragma**

適用可能エレメント

ローカルに適用されます。

適用ルール

合成でコードの一部を有効または無効にするよう指定します。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
-- synthesis translate_off
...code not synthesized...
-- synthesis translate_on
```

Verilog の構文例

HDL のメタ コメントとして使用できます。この Verilog 構文は、標準的なメタ コメント構文とは異なります。

```
// synthesis translate_off
...code not synthesized...
// synthesis translate_on
```

Verilog インクルード ディレクトリ

- ・ Verilog の 'include ディレクトリの指定 (-vlgincdir) コマンド ライン オプションには、次の特徴があります。
 - 'include と共に使用されます。
 - 'include 文で参照されるファイルをパーサーが見つけやすくなります。
- ・ 'include 文によりファイルが参照されると、XST では次の順序でさまざまなディレクトリが検索されます。
 - 現在のディレクトリ
 - 'include を含む Verilog ファイルのディレクトリ
 - .prj ファイルのディレクトリ
 - -vlgincdir オプションで参照されるディレクトリ

適用可能エレメント

ディレクトリに適用されます。

適用ルール

ありません。

制約値

指定できる値はディレクトリ名です。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-vlgincdir {directory_path [directory_path]}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Verilog Include Directories]

この制約を表示するには、[Edit] → [Preferences] → [Processes] をクリックし、[Property display level] を [Advanced] に設定します。

Verilog マクロ

Verilog マクロ (`-define`) コマンドライン オプションには、次の特徴があります。

- ・ Verilog マクロを定義または再定義します。
- ・ ソースコードを変更しなくてもデザイン コンフィギュレーションを簡単に修正できます。
- ・ IP コアの生成やフロー テストで使用できます。定義されたマクロがデザインで使用されていない場合は、何のメッセージも表示されません。

適用可能エレメント

グローバルに適用します。

適用ルール

ありません。

制約値

- ・ *name* はマクロ名
- ・ *value* はマクロ テキスト

デフォルトでは何も定義されていません。

```
-define {}
```

構文ルール

- ・ マクロの値は必須ではありません。
- ・ {} 内に値を挿入します。
- ・ 各値はスペースで区切ります。
- ・ マクロ テキストは、二重引用符 (") で囲むか、そのまま指定します。ただし、マクロ テキストにスペースが含まれる場合は、必ず二重引用符を使用してください。

```
-define {macro1=Xilinx macro2="Xilinx Virtex6"}
```

- ・ ISE® Design Suite で値を指定する場合は、{} は使用しないでください。

```
acro1=Xilinx macro2="Xilinx Virtex6"
```

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンドライン構文例

run コマンドでグローバルに設定します。

```
-define {name[=value] name[=value] -}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Verilog Macros]

作業ディレクトリ

作業ディレクトリ (**-xsthdpdir**) コマンドライン オプションでは、ライブラリ マップ ファイルで場所が指定されていない場合に、コンパイルされたファイルを保存する場所を定義します。

- ISE® Design Suite
[Process Properties] ダイアログ ボックス → [Synthesis Options] → [VHDL Work Directory]

- コマンドライン モード

```
set -xsthdpdir directory
```

具体例

- 3 人のユーザーが同じプロジェクトを作業しています。
- あらかじめコンパイルされた shlib という標準ライブラリを共有しています。
- このライブラリには、プロジェクトで使用するマクロ ブロックが含まれています。
- 各ユーザーは、それぞれローカルにも作業ライブラリを持っています。
- ユーザー 3 はこれをプロジェクト ディレクトリ以外の場所 (c:\temp) に保存しています。
- ユーザー 1 と 2 は、上記以外にライブラリ lib12 を共有していますが、ユーザー 3 とは共有していません。

ユーザー 1

```
Mapping file:  
schlib=z:\sharedlibs\shlib  
lib12=z:\userlibs\lib12
```

ユーザー 2

```
Mapping file:  
schlib=z:\sharedlibs\shlib  
lib12=z:\userlibs\lib12
```

ユーザー 3

```
Mapping file:  
schlib=z:\sharedlibs\shlib
```

ユーザー 3 は、次も設定する必要があります。

```
XSTHDPDIR = c:\temp
```

適用可能エレメント

ディレクトリに適用されます。

適用ルール

ありません。

制約値

指定できる値はディレクトリ名です。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
set -xsthdpdir directory
```

このコマンドで指定できるパスは 1 つのみです。使用するディレクトリを指定します。

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [VHDL Work Directory]

この制約を表示するには、[Edit] → [Preferences] → [Processes] をクリックし、[Property display level] を [Advanced] に設定します。

HDL 制約

この章では、XST の HDL 制約について説明しています。

この章には、ほとんどの制約の次の情報を含めます。

- ・ 制約の説明
- ・ 適用可能エレメント
- ・ 適用ルール
- ・ 制約値
- ・ 構文例

FSM 自動抽出

FSM 自動抽出 (FSM_EXTRACT) 制約には、次の特徴があります。

- ・ 有限ステート マシンの抽出、および特定の合成の最適化オプションを有効にできます。
- ・ [\[FSM Encoding Algorithm\]](#) および [\[FSM Flip-Flop Type\]](#) の値を設定するには、このオプションがオンになっている必要があります。

適用可能エレメント

グローバルに適用するか、またはエンティティ、モジュール、信号に適用できます。

適用ルール

設定したエンティティ、モジュール、または信号に適用されます。

制約値

- ・ yes [または true (XCF)] (デフォルト)
- ・ no [または false (XCF)]

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute fsm_extract: string;
```

次のように指定します。

```
attribute fsm_extract of {entity_name|signal_name}:{entity|signal  
is "{yes|no}";
```

Verilog の構文例

次をモジュールまたは信号宣言の直前に入力します。

```
(* fsm_extract = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" fsm_extract={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"
```

```
NET "signal_name" fsm_extract={yes|no|true|false};
```

```
END;
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

-fsm_extract {yes|no}*

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [FSM Encoding Algorithm]

- ・ **FSM エンコーディング アルゴリズム (FSM_ENCODING)** が none に設定され、**-fsm_extract** を no に設定されると、**-fsm_encoding** の設定は合成には関係なくなります。
- ・ その他の場合は、**-fsm_extract** は yes に設定され、**-fsm_encoding** は選択した値に設定されます。

列挙型エンコード手法

ENUM_ENCODING (列挙型エンコード手法) 制約には、次のような特徴があります。

- ・ VHDL 列挙型に適用するエンコード手法を選択できます。
- ・ 列挙型で VHDL 制約としてのみ指定できます。
- ・ ステートレジスタの列挙型を使用して FSM (有限ステートマシン) のエンコード方法を指定できます。
- ・ ステートレジスタの [FSM エンコード方法の指定 \(FSM_ENCODING\)](#) を user に設定します。

適用可能エレメント

- ・ 信号または型に適用されます。
- ・ ENUM_ENCODING は外部デザインのインターフェイスを保持する必要があるので、ポートに設定された場合は XST で無視されます。

適用ルール

設定されたタイプまたは信号に適用されます。

制約値

属性値は、空白で区切られたバイナリコードを含む文字列です。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次の例に示すように、VHDL 制約として列挙型に指定します。

```
...
architecture behavior of example is
type statetype is (ST0, ST1, ST2, ST3);
attribute enum_encoding : string;
attribute enum_encoding of statetype : type is "001 010 100 111";
signal statel : statetype;
signal state2 : statetype;
begin
...
```

XCF の構文例

```
BEGIN MODEL "entity_name"

NET "signal_name" enum_encoding="string";

END;
```


等価レジスタの削除

等価レジスタの削除 (EQUIVALENT_REGISTER_REMOVAL) 制約は、RTL レベルで記述された等価レジスタの削除ができます。

- ・ ザイリンクス プリミティブ ライブラリからインスタンス化された等価フリップフロップは削除されません。
- ・ 等価フリップフロップを削除すると、デザインがデバイスにフィットする可能性が上がります。

適用可能エレメント

グローバルに適用するか、またはエンティティ、モジュール、信号に適用できます。

適用ルール

同等フリップフロップおよび定数入力を使用したフリップフロップを削除します。

制約値

- ・ yes [または true (XCF)] (デフォルト)
フリップフロップの最適化が有効です。
- ・ no [または false (XCF)]
フリップフロップの最適化が無効です。

フリップフロップの最適化には時間がかかります。高速処理が必要な場合は、no に設定してください。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute equivalent_register_removal: string;
```

次のように指定します。

```
attribute equivalent_register_removal of  
{entity_name|signal_name}: {signal|entity} is "{yes|no}";
```

Verilog の構文例

次をモジュールまたは信号宣言の直前に入力します。

```
(* equivalent_register_removal = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name"  
equivalent_register_removal={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
  
NET "signal_name"  
equivalent_register_removal={yes|no|true|false};  
  
END;
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-equivalent_register_removal {yes|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Xilinx-Specific Options] → [Equivalent Register Removal]

FSM エンコード方法の指定

- ・ FSM_ENCODING 制約を使用すると、FSM のコーディング手法を選択できます。
- ・ このプロパティを設定するには、FSM 自動抽出 (FSM_EXTRACTION) をオンにしておく必要があります。

適用可能エレメント

グローバルに適用するか、またはエンティティ、モジュール、信号に適用できます。

適用ルール

設定したエンティティ、モジュール、または信号に適用されます。

制約値

- ・ auto (デフォルト)
各ステート マシンに最適なコーディング手法が自動的に選択されます。
- ・ one-hot
- ・ compact
- ・ sequential
- ・ gray
- ・ johnson
- ・ speed1
- ・ user

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute fsm_encoding: string;
```

次のように指定します。

```
attribute fsm_encoding of  
{entity_name|signal_name}:{entity|signal} is  
"{auto|one-hot|compact|sequential|gray|johnson|speed1|user}";
```

Verilog の構文例

次をモジュールまたは信号宣言の直前に入力します。

```
(* fsm_encoding = "{auto|one-hot  
|compact|sequential|gray|johnson|speed1|user}" *)
```

XCF の構文例 1

```
MODEL "entity_name" fsm_encoding={auto|one-hot  
|compact|sequential|gray|johnson|speed1|user};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
  
NET "signal_name" fsm_encoding={auto|one-hot  
|compact|sequential|gray|johnson|speed1|user};  
  
END;
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-fsm_encoding  
{auto|one-hot|compact|sequential|gray|johnson|speed1|user}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [FSM Encoding Algorithm]

- ・ [None] を選択すると、-fsm_extract は no に設定され、-fsm_encoding の設定は合成には関係なくなります。
- ・ その他の場合は、-fsm_extract は yes に設定され、-fsm_encoding はメニューで選択した値に設定されます。

詳細は、「[FSM 自動抽出](#)」を参照してください。

MUX の最小サイズ

注意：この制約は使用する前に注意が必要です。

MUX の最小サイズ (MUX_MIN_SIZE) 制約には、次のような特徴があります。

- ・ XST で推論されるマルチプレクサー マクロの最小サイズを制御します。
- ・ 1 よりも大きな整数値を指定できます。デフォルトは 2 です。

多重化されたデータ入力数

「サイズ」は、マルチプレクサーを通るデータ入力の数です。セレクト入力はカウントしません。

マルチプレクサー	サイズ
2-to-1 Multiplexer	2
16-to-1 Multiplexer	16

選択データの幅

サイズは、選択したデータの幅とは無関係です。

マルチプレクサー	サイズ
1 ビット幅の 8:1 マルチプレクサー	8
16 ビット幅の 8:1 マルチプレクサー	8

2:1 マルチプレクサー マクロ

XST では、2:1 マルチプレクサー マクロが自動的に推論されます。

- ・ 2:1 マルチプレクサーを明示的推論すると、デバイス使用率に好影響があったり、悪影響があったりします。
 - デバイス使用率に問題がない場合、この制約は使用しないでください。
 - デバイス使用率に問題がある場合は、使用するとデバイス使用率が改善されることがあります。

注記：デバイス使用率に問題がある原因としては、推論された 2:1 マルチプレクサーが多いということが挙げられます。2:1 マルチプレクサーの推論をディスエーブルにするには、グローバルまたはその問題となっている特定ブロックのいずれかに対する MUX_MIN_SIZE の値に 3 を指定します。

- ・ MUX_MIN_SIZE を使用することで 2 より大きなサイズのマルチプレクサーの推論が回避はできますが、それによる利点があるかどうかは確実ではありません。MUX_MIN_SIZE に 2 より大きい値を使用する場合には、十分な注意が必要です。

適用可能エレメント

グローバルに、または VHDL エンティティまたは Verilog モジュールに適用します。

適用ルール

エンティティまたは Verilog モジュールに適用します。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute mux_min_size: string;
```

次のように指定します。

```
attribute mux_min_size of entity_name : entity is "integer";
```

Verilog の構文例

次をモジュール宣言の直前に入力します。

```
(* mux_min_size= "integer" *)
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-mux_min_size integer
```

この制約は ISE® Design Suite のデフォルトの XST オプションには含まれません。

リソース共有

リソース共有 (RESOURCE_SHARING) 制約を使用すると、[数値演算子](#)のリソース共有を有効または無効にします。

適用可能エレメント

グローバルに適用するか、デザイン エレメントに適用します。

適用ルール

設定したエンティティまたはモジュールに適用されます。

制約値

- yes [または true (XCF)] (デフォルト)
- no [または false (XCF)]

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「[はじめに](#)」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute resource_sharing: string;
```

次のように指定します。

```
attribute resource_sharing of entity_name: entity is "{yes|no}";
```

Verilog の構文例

次をモジュール宣言またはインスタンス化の直前に入力します。

```
attribute resource_sharing of entity_name: entity is "{yes|no}";
```

XCF の構文例 1

```
MODEL "entity_name" resource_sharing={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"
```

```
NET "signal_name" resource_sharing={yes|no|true|false};
```

```
END;
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-resource_sharing {yes|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [Resource Sharing]

セーフ インプリメンテーション

セーフ インプリメンテーション (SAFE_IMPLEMENTATION) 制約を使用すると、有限ステートマシン (FSM) をセーフ インプリメンテーション モードでインプリメントできます。

- ・ FSM が無効な状態になると、XST では追加ロジックを生成して、FSM を有効な状態 (リカバリ ステート) にします。
 - デフォルトでは、リカバリ ステートとして reset が選択されます。
 - FSM に初期信号が含まれていない場合は、power-up が選択されます。
 - リカバリ ステートは、[セーフ リカバリ ステート](#)制約を使用して手動で定義します。
- ・ セーフ インプリメンテーションをオンにするには、次の手順に従ってください。
 - ISE® Design Suite
[Process Properties] ダイアログ ボックス → [HDL Options] → [Safe Implementation] をクリックします。
 - Hardware Description Language (HDL)
ステートレジスタを表す階層ブロックまたは信号に SAFE_IMPLEMENTATION 制約を設定します。

適用可能エレメント

デザイン全体 (XST コマンド ラインを使用)、特定ブロック (エンティティ、アーキテクチャ、コンポーネント)、または信号に適用されます。

適用ルール

設定したエンティティ、コンポーネント、モジュール、信号、インスタンスに適用されます。

制約値

- ・ yes [または true (XCF)]
- ・ no [または false (XCF)] (デフォルト)

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute safe_implementation: string;
```

次のように指定します。

```
attribute safe_implementation of  
entity_name|component_name|signal_name):  
entity|component|signal is "{yes|no}";
```

Verilog の構文例

次をモジュールまたは信号宣言の直前に入力します。

```
(* safe_implementation = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" safe_implementation={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
  
NET "signal_name"safe_implementation="{yes|no|true|false} ;  
  
END;
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-safe_implementation {yes|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [Safe Implementation]

セーフ リカバリ ステート

セーフ リカバリ ステート (SAFE_RECOVERY_STATE) 制約を使用すると、有限ステート マシン (FSM) をセーフ インプリメンテーション モードでインプリメントする際に使用するリカバリ ステートを定義できます。

- ・ FSM が無効な状態になると、XST では追加ロジックを使用して、FSM を有効な状態にします。
- ・ FSMをセーフ モードでインプリメントすると、FSMの普通のビヘイビアには含まれないコードを集めて、無効なコードとして処理します。
- ・ XST では、FSM を次のステートと同時に戻すロジックが使用されます。
 - 既知のステート
 - リセット ステート
 - 電源投入ステート
 - SAFE_RECOVERY_STATE を使用して指定したステート

適用可能エレメント

ステート レジスタを表す信号に適用されます。

適用ルール

設定された信号に適用されます。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute safe_recovery_state: string;
```

次のように指定します。

```
attribute safe_recovery_state of {signal_name}:{signal} is  
"<value>";
```

Verilog の構文例

次を信号宣言の直前に入力します。

```
(* safe_recovery_state = "<value>" *)*
```

XCF の構文例

```
BEGIN MODEL "entity_name"  
  
NET "signal_name" safe_recovery_state="<value>";  
  
END;
```

FPGA 制約（タイミング制約以外）

この章では、タイミング制約以外の FPGA 制約について説明します。

この章には、ほとんどの制約の次の情報を含めます。

- ・ 制約の説明
- ・ 適用可能エレメント
- ・ 適用ルール
- ・ 制約値
- ・ 構文例

多くの制約は、次のように適用できます。

- ・ デザイン全体またはモデルにグローバルに適用
- ・ 個別の信号、ネット、インスタンスにローカルに適用

非同期から同期への変換

注意： この制約で非同期から同期へ変換することによって、どのような影響があるか注意してください。

この制約を使用すると、非同期 set および reset 信号を同期信号に置き換えることができます。

非同期から同期への変換

この制約には、次のような特徴があります。

- ・ 推論済みのシーケンシャル エLEMENTのみに適用されます。
- ・ インスタンス化されたフリップフロップおよびラッチには適用されません。
- ・ オンザフライで実行されます。
- ・ 合成後のネットリストに反映されます。
- ・ HDL ソース コードは変更されません。

セットおよびリセット

- ・ ブロック RAM および DSP ブロックなどのザイリンクス デバイス リソースの set および reset は元々同期信号です。
- ・ コーディング方法で set および reset 信号を非同期に記述する必要がある場合は、可能性を最大に高めるため、これらのリソースを使用する必要のないこともあります。
- ・ 非同期から同期への変換を実行させると、HDL ソース コードの順次エLEMENTの記述を変更せずに、これらのリソースを生かすことができます。
- ・ レジスタをより活用することで、次が可能になります。
 - デバイス使用率の改善
 - 回路パフォーマンスの増加
 - 電力削減

合成後のネットリスト

- ・ 非同期から同期への変換により、合成後のネットリストは合成前の HDL 記述とは理論的に変わります。
- ・ ただし、合成後のネットリストは次の場合同じ機能になります。
 - 変換で記述した非同期セットおよびリセットが実際に使用されない場合
 - 非同期セットおよびリセットが同期ソースから派生している場合

HDL 記述の変更

非同期から同期への変換によりデザイン目標を達成したら、次のために HDL 記述を変更するかどうか判断します。

- ・ 予測通りの回路動作にするために同期 set および reset 信号を使用する
- ・ 設計検証をしやすくする

推薦事項

次の事項に従うことをお勧めします。

- ・ 非同期から同期へ変換するこの制約を生かすには、タイミング シミュレーションを実行します。
- ・ HDL ソース コードで同期 set および reset 信号を記述します。

適用可能エレメント

グローバルに適用します。

適用ルール

ありません。

制約値

- ・ **yes**
- ・ **no** (デフォルト)

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-async_to_sync {yes|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [Asynchronous to Synchronous]

自動 BRAM パッキング

この制約 (AUTO_BRAM_PACKING) を使用すると、2 つの小型ブロック RAM をデュアル ポートブロック RAM として 1 つのブロック RAM プリミティブにパッキングできます。

- ・ XST ではブロック RAM が同じ階層レベルにある場合にのみパッキングします。
- ・ この制約はデフォルトではオフになっています。

適用可能エレメント

グローバルに適用します。

適用ルール

ありません。

制約値

- ・ **yes**
- ・ **no** (デフォルト)

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-auto_bram_packing {yes|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Automatic BRAM Packing]

ブロック RAM Read-First インプリメンテーション

ブロック RAM Read-First インプリメンテーション (RDADDR_COLLISION_HWCONFIG) 制約では、read-first 同期で記述されるブロック RAM のインプリメンテーションが制御されます。

ブロック RAM Read-First インプリメンテーションには、次のような特徴があります。

- ・ Virtex®-6 デバイスにのみ適用できます。
- ・ 次の場合は無視されます。
 - Virtex-6 デバイス以外のデバイス ファミリをターゲットにしている場合
 - 記述された read-write 同期が read-first ではない場合
- ・ 次に設定できます。
 - インスタンシエート済みブロック RAM プリミティブ
 - 推論済み RAM
- ・ 推論済み RAM の場合、記述されたメモリを read-first 同期にするかどうかは XST では指定できません。これには、HDL コードで正しく設定する必要があります。
- ・ 次からオプションとして指定できません。
 - ISE® Design Suite
 - [コマンド ライン モード](#)

詳細は、「[ブロック RAM の読み出し/書き込みの同期](#)」を参照してください。

適用可能エレメント

- ・ 次を通してローカルに適用されます。
 - VHDL 属性
 - Verilog 属性
 - XCF 制約
- ・ 次に適用されます。
 - ブロック：
 - ◆ エンティティ
 - ◆ アーキテクチャ
 - ◆ コンポーネント
 - RAM を記述する信号

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

制約値

- delayed_write
 ブロック RAM はメモリ破損しないようにコンフィギュレーションされます。競合は回避されますが、write-first および no-change 同期に比べると、パフォーマンスが落ちます。
- performance
 read-first モードのパフォーマンスを最大にします。パフォーマンスは、write-first および no-change モードと同等になりますが、メモリ破損が発生しないように注意する必要があります。

デフォルト値

RAM が推論されている場合、デフォルト値は RAM ポートの数によって異なります。

ポート	デフォルト値	メモ
シングル ポート	performance	RAM がデュアル ポートの場合にのみメモリ破損が発生する可能性があります。このため、メモリがシングル ポートの場合は performance モードを問題なく使用できます。
デュアル ポート	delayed_write	

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute RDADDR_COLLISION_HWCONFIG: string;
```

次のように指定します。

```
attribute RDADDR_COLLISION_HWCONFIG of
"entity_name|component_name|signal_name":{entity|component|signal}
is "{delayed_write|performance}";
```

Verilog の構文例

インスタンス、モジュール、または信号宣言の直前に入力します。

```
(*RDADDR_COLLISION_HWCONFIG = "{delayed_write|performance}" *)
```

XCF の構文例 1

```
MODEL "entity_name" RDADDR_COLLISION_HWCONFIG
={delayed_write|performance};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"
```

```
NET "signal_name" RDADDR_COLLISION_HWCONFIG
={delayed_write|performance};

END;
```

BRAM 使用率

BRAM 使用率 (BRAM_UTILIZATION_RATIO) 制約を使用すると、合成中に XST で処理されるブロック RAM コンポーネントの数を制限できます。

- ・ ブロック RAM コンポーネントは次で作成されることがあります。
 - ブロック RAM の推論プロセス
 - インスタンシエーションとブロック RAM マップ最適化
- ・ ロジックの RTL 記述を別のブロックに分けてから、XST でこのロジックがブロック RAM にマップされるようにします。

詳細は、「[ブロック RAM へのロジックのマップ](#)」を参照してください。

- ・ インスタンシエートされたブロック RAM は使用可能なブロック RAM リソースの第一候補として認識されます。
 - この推論された RAM が残りのブロック RAM リソースに配置されます。
 - インスタンシエートされたブロック RAM の数が使用可能なリソース数を上回ってしまう場合、XST でインスタンシエーションが修正されず、それらはブロック RAM スライスとしてはインプリメントされません。
 - 特定の RAM をブロック RAM としてインプリメントした場合も、同じビヘイビアになります。
 - リソースがない場合は、ブロック RAM リソースの数が超えていても、ユーザー制約が優先されます。
- ・ ユーザーの指定したブロック RAM の数がターゲット デバイスの使用可能なブロック RAM リソース数を上回る場合は、XST で次が実行されます。
 - 警告メッセージが表示されます。
 - 使用可能なブロック RAM リソースのみが合成に使用されます。
- ・ 自動的にブロック RAM リソースが管理されないようにするには、値に -1 を指定します。これにより、特定デザインで推論されるブロック RAM の数を確認できます。
- ・ ブロック RAM 数がデバイスで使用可能なブロック RAM 数を大幅に上回っている場合 (何百個も上回る場合)、合成にかなり時間がかかります。これは、フィットできないブロック RAM がすべて分散 RAM に変換されると、デザインが複雑になるためです。

適用可能エレメント

グローバルに適用します。

適用ルール

ありません。

制約値

- ・ integer の範囲は -1 ～ 100 です。
- ・ % ではパーセントの値を、# ではブロック RAM の絶対数を示します。
- ・ 整数値と % または # 文字の間にはスペースを入れないでください。
- ・ % と # のどちらも記述されない場合は、% であると仮定されます。
- ・ デフォルト値は 100 です (XST は使用可能なブロック RAM リソースを 100% 使い切ります)。
- ・ 値が -1 の場合：
 - 自動ブロック RAM リソース管理がオフになります。
 - XST で潜在的に推論されるブロック RAM 数を入手しやすくなります。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンド ラインの構文例

run コマンドでグローバルに設定します。

```
-bram_utilization_ratio <integer>[%|#]
```

XST コマンド ライン構文例 1

```
-bram_utilization_ratio 50
```

ターゲット デバイスでブロック RAM の 50% が使用されます。

XST コマンド ラインの構文例 2

```
-bram_utilization_ratio 50%
```

ターゲット デバイスでブロック RAM の 50% が使用されます。

XST コマンド ラインの構文例 3

```
-bram_utilization_ratio 50#
```

ブロック RAM 50 個

- ・ 整数値と % および # 文字の間にはスペースを入れないでください。
- ・ 場合によっては、自動的にブロック RAM リソースが管理されないようにすることもできます。
たとえば、特定デザインに対して XST で推論されるブロック RAM 数を確認する場合などは、この DSP リソース管理オプションをオフにできます。
- ・ オフにするには、-1 または負の数を制約値として指定します。

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックスを表示して、[Synthesis Options] → [BRAM Utilization Ratio] をクリックします。

ISE® Design Suite ではこのオプションの値を % として定義できます。ブロック RAM 数は絶対値では定義できません。

バッファ タイプ (BUFFER_TYPE)

バッファ タイプ (BUFFER_TYPE) 制約では、I/O ポートまたは内部ネットに挿入するバッファのタイプを指定します。

適用可能エレメント

信号に適用されます。

適用ルール

設定された信号に適用されます。

制約値

- ・ ibufg
- ・ bufg
- ・ bufgp
- ・ bufh
- ・ bufr
- bufio
- ibuf
- obuf
- buf
- none

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute buffer_type: string;
```

次のように指定します。

```
attribute buffer_type of signal_name: signal is  
"{ibufg|bufg|bufgp|bufh|bufr|bufio|ibuf|obuf|buf|none}";
```

Verilog の構文例

次を信号宣言の直前に入力します。

```
(* buffer_type = "{ibufg|bufg|bufgp|bufh|bufr|bufio|ibuf|obuf|buf|none}" *)
```

XCF の構文例

```
BEGIN MODEL "entity_name"  
NET "signal_name" buffer_type={ibufg|bufg|bufgp|bufh|bufr|bufio|ibuf|obuf|buf|none};  
END;
```

トライステートからロジックへの変換

デバイスによっては、内部トライステートがサポートされません。XST は、TRISTATE2LOGIC 制約を使用して、これらのデバイスの内部トライステートを同等のロジックに変換します。

- ・ 同等のロジックは、周辺のロジックと組み合わせて最適化ができます。
- ・ 内部トライステートを同等ロジックに変換すると、次が可能になることがあります。
 - － スピードの向上
 - － エリア最適化の改善
- ・ 内部トライステートを同等ロジックに変換すると、通常はエリアが増加します。最適化目標を area に設定している場合は、TRISTATE2LOGIC を no に設定してください。

制限事項

- ・ 同等のロジックに変換されるのは、内部トライステートのみです。出力パッドに接続された最上位モジュールのトライステートは保持されます。
- ・ インクリメンタル合成がアクティブになっている場合は、内部トライステートは同等ロジックに変換されません。
- ・ XST では、次の場合、内部トライステートを同等ロジックに置換できません。
 - － トライステートが次に接続される場合：
 - ◆ **ブラック ボックス**
 - ◆ ブロックの階層が保持される場合のブロックの出力
 - ◆ 最上位レベルの出力
 - － この制約は、次のいずれかでは no に設定されます。
 - ◆ トライステートが配置されたブロック
 - ◆ トライステートが接続された信号

適用可能エレメント

デザイン全体 (XST コマンドラインを使用)、特定ブロック (エンティティ、アーキテクチャ、コンポーネント)、または信号に適用されます。

適用ルール

設定したエンティティ、コンポーネント、モジュール、信号、インスタンスに適用されます。

制約値

- ・ yes [または true (XCF)] (デフォルト)
- ・ no [または false (XCF)]

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute tristate2logic: string;
```

次のように指定します。

```
attribute tristate2logic of  
{entity_name|component_name|signal_name}:{entity|component|signal}  
is "{yes|no}";
```

Verilog の構文例

次をモジュールまたは信号宣言の直前に入力します。

```
(* tristate2logic = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" tristate2logic={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" tristate2logic={yes|no|true|false};  
END;
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-tristate2logic {yes|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Xilinx-Specific Options] → [Convert Tristates to Logic]

コアの検索ディレクトリ

コアの検索ディレクトリ (-sd) コマンドライン オプションを使用すると、デフォルトのディレクトリ以外にもコアを検索するディレクトリを指定できます。

- ・ デフォルトでは、-ifn オプションで指定されたディレクトリでコアが検索されます。
- ・ コアを含むディレクトリのみをリストし、各コア ファイルはリストしないようにします。
- ・ コア ディレクトリを絶対パスか相対パスで指定します。

適用可能エレメント

グローバルに適用します。

適用ルール

ありません。

制約値

- ・ 値は、1 つのディレクトリ名か、複数のディレクトリ名のリストになります。
 - ディレクトリのリストは { } で囲みます。
 - 1 つのディレクトリのみを指定する場合は、必要ありません。
 - ディレクトリ名はスペースで区切ります。
- ・ ディレクトリ名にスペースを含めるのは推奨されません。
 - 次の例のように二重引用符 (") で囲めば、スペースがあっても検索リストに含めることができます。

```
-sd {"/mydir1/mysubdir1" "/mydir2" "/mydir3/mysubdir  
with space" }
```
 - 詳細は、「[コマンドライン モードでのスペースを含む名前](#)」を参照してください。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「[はじめに](#)」の構文例を参照してください。

XST コマンドライン構文例

run コマンドでグローバルに設定します。

```
-sd {directory_path [directory_path]}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Cores Search Directory]

DSP の使用率

DSP 使用率 (DSP_UTILIZATION_RATIO) 制約を使用すると、推論済みファンクションをインプリメントするために XST で使用される DSP ブロックの数を制限できます。

- ・ XST は使用可能なリソースの制限内で DSP ブロックを推論します。
- ・ この制約により、XST でこれらのリソースすべてが使用されるのを回避できます。

DSP リソースのバジェット

- ・ この制約は、共同ワークフローで通常使用されます。
 - さまざまなコンポーネントは最初別々にデザインされてから最終プロジェクトに統合されます。
 - この制約の値にしたがって、各コンポーネントの DSP リソースが割り当てられます。
- ・ この制約は、次のいずれかを定義します。
 - DSP スライスの絶対数
 - デバイスで使用可能なリソース総計のパーセンテージ
- ・ デフォルトでは、100% の選択デバイスで使用可能な DSP リソースに設定されています。
- ・ デバイスで使用可能な DSP リソースを超える絶対数またはパーセンテージを定義すると、XST で使用可能な数より多いリソースは使用できないことを示すメッセージが表示されます。
- ・ インスタンス化された DSP プリミティブが最初に使用されます。XST は DSP_UTILIZATION_RATIO で定義されたトータル バジレットから対応する量を分配します。この後、残りのリソースを使用して推論済みファンクションをインプリメントします。
- ・ 定義されたバジレットは、次の場合に超過することがあります。
 - インスタンス化された DSP ブロックの数が定義されたバジレットよりも多い場合。XST では常に DSP インスタンス化がすべて認識されるので、選択デバイスがインスタンス化された DSP ブロックすべてに適応するようにする必要があります。
 - 推論済みマクロの DSP を強制的にインプリメントするために、[DSP ブロックの使用 \(USE_DSP48\)](#) 制約の値を yes にした場合
- ・ DSP ブロックの使用 (USE_DSP48) の値を yes にした場合、XST は DSP_UTILIZATION_RATIO で定義された最大の DSP アロケーション (割合) と選択デバイスで実際に使用可能な DSP リソース量の両方を無視します。このため、デザインがデバイスにフィットしないこともあります。DSP_UTILIZATION_RATIO は、DSP ブロックの使用 (USE_DSP48) の値を auto および automax にした場合に最も効果的です。

自動 DSP リソース管理機能の無効化

自動的にリソースが管理されないようにするには、DSP_UTILIZATION_RATIO を -1 (または負の値) に指定します。たとえば、特定デザインに対して XST で推論される DSP の数を確認する場合などは、この DSP リソース管理オプションをオフにすることもできます。

適用可能エレメント

グローバルに適用します。

適用ルール

ありません。

制約値

- ・ `<integer>` の範囲は、% が使用されるか、% と # のどちらも削除される場合は -1 ~ 100 になります。
- ・ 合計スライスの割合は、次のように指定します。

```
-dsp_utilization_ratio 50
```

または

```
-dsp_utilization_ratio 50%
```
- ・ スライスの絶対値は、次のように指定します。

```
-dsp_utilization_ratio 22.68kg
```
- ・ 整数値と % および # 文字の間にはスペースを入れないでください。
- ・ デフォルトは % です。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-dsp_utilization_ratio number[%|#]
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [DSP Utilization Ratio]

ISE® Design Suite ではこのオプションの値を % として定義できます。スライスの絶対値は指定できません。

BUFGCE の抽出

BUFGCE の抽出 (BUFGCE) 制約を使用すると、BUFGMUX プリミティブを推論して BUFGMUX の機能をインプリメントできます。

- ・ この動作によって、ワイヤ数が低減されます。
- ・ クロック信号およびクロック イネーブル信号は、1 本のワイヤで N 個の順序コンポーネントに送られます。
- ・ この制約は、プライマリ クロック信号に設定する必要があります。
- ・ BUFGCE は、HDL コードで設定できます。

適用可能エレメント

クロック信号に適用されます。

適用ルール

設定された信号に適用されます。

制約値

- ・ yes [または true (XCF)]
 - bufgce=yes に設定すると、BUFGMUX の機能が可能な限りインプリメントされます。
 - このとき、すべてのフリップフロップで同じクロック イネーブル信号が使用されている必要があります。
- ・ no [または false (XCF)]

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute bufgce : string;
```

次のように指定します。

```
attribute bufgce of signal_name: signal is "{yes|no}";
```

Verilog の構文例

次を信号宣言の直前に入力します。

```
(* bufgce = "{yes|no}" *)
```

XCF の構文例 1

```
BEGIN MODEL "entity_name"
```

```
NET "primary_clock_signal" bufgce={yes|no|true|false};
```

```
END;
```

FSM スタイル

FSM スタイル (FSM_STYLE) 制約を使用すると、大型の FSM をブロック RAM リソースにコンパクトで高速にインプリメントできます。

- ・ これはグローバルにも、またはローカルにも設定できます。
- ・ LUT (デフォルト) ではなく、ブロック RAM リソースを使用するように指定して FSM をインプリメントします。

適用可能エレメント

グローバルに適用するか、またはエンティティ、モジュール、信号に適用できます。

適用ルール

設定したエンティティ、モジュール、または信号に適用されます。

制約値

- ・ lut (デフォルト)
- ・ bram

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute fsm_style: string;
```

次のように指定します。

```
attribute fsm_style of {entity_name|signal_name}:{entity|signal}  
is "{lut|bram}";
```

Verilog の構文例

インスタンス、モジュール、または信号宣言の直前に入力します。

```
(* fsm_style = "{lut|bram}" *)
```

XCF の構文例 1

```
MODEL "entity_name" fsm_style = {lut|bram};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
  
NET "signal_name" fsm_style = {lut|bram};  
  
END;
```

XCF の構文例 3

```
BEGIN MODEL "entity_name"  
  
INST "instance_name" fsm_style = {lut|bram};  
  
END;
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [FSM Style]

LUT の結合

LUT の結合 (LC) 制約を使用すると、共通の入力を持つ LUT ペアを 1 つのデュアル出力の LUT6 にまとめて、エリアを削減できます。

この最適化プロセスでは、次が可能になることがあります。

- ・ デザイン エリアの改善
- ・ デザイン速度の削減

適用可能エレメント

グローバルに適用します。

適用ルール

ありません。

制約値

- ・ auto (デフォルト)
XST でエリアとスピード間のトレードオフが考慮されます。
- ・ area
できるだけ小さいエリアのインプリメンテーションにするため、LUT の結合が最大限に実行されます。
- ・ off
LUT は結合されません。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-lc {auto|area|off}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [LUT Combining]

単一 LUT へのエンティティのマップ

単一 LUT へのエンティティのマップ (LUT_MAP) 制約を使用すると、1 つのブロックを 1 つの LUT にマップするように指定できます。

- ・ RTL レベルで記述された機能が 1 つの LUT にフィットしない場合は、エラー メッセージが表示されます。
- ・ XST では、Synplify の xc_map 制約が認識されます。

詳細は、「[LUT へのロジックのマップ](#)」を参照してください。

UNISIM ライブラリの使用

LUT コンポーネントを直接 HDL ソースコードにインスタンス化するには、UNISIM ライブラリを使用します。

- ・ LUT のファンクションを指定するには、LUT のインスタンスに INIT 制約を設定します。
- ・ インスタンス化した LUT またはレジスタを特定のスライスに配置する場合は、同じインスタンスに **RLOC** 制約を設定します。

HDL ソースコードでのファンクションの記述

1 つの LUT にマップするファンクションを HDL ソースコードで記述します。

1. このファンクションは別のブロックに記述します。
2. このブロックに LUT_MAP 制約を設定すると、このブロックが 1 つの LUT にマップされます。
3. LUT の INIT 値は XST により算出されます。
4. 最適化中はこの LUT が保持されます。

適用可能エレメント

エンティティまたはモジュールに適用します。

適用ルール

設定したエンティティまたはモジュールに適用されます。

制約値

- ・ yes [または true (XCF)]
- ・ no [または false (XCF)]

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「[はじめに](#)」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute lut_map: string;
```

次のように指定します。

```
attribute lut_map of entity_name: entity is "{yes|no}";
```

Verilog の構文例

次をモジュール宣言またはインスタンスエーションの直前に入力します。

```
(* lut_map = "{yes|no}" *)
```

XCF の構文例

```
MODEL "entity_name" lut_map={yes|no|true|false};
```

BRAM へのロジックへのマップ

BRAM へのロジックのマップ (BRAM_MAP) 制約では、階層ブロック全体をブロック RAM リソースにマップできます。

- ・ BRAM_MAP は、グローバルに、またはローカルに設定できます。
- ・ 詳細は、「[ブロック RAM へのロジックのマップ](#)」を参照してください。

適用可能エレメント

ブロック RAM コンポーネントに適用されます。

適用ルール

- ・ RAM にマップするロジック (出力レジスタを含む) は、異なる階層レベルに指定する必要があります。
- ・ ロジックが 1 つのブロック RAM にフィットしない場合、そのロジックはマップされません。
- ・ エンティティの一部だけでなく、全体がフィットすることを確認してください。
- ・ BRAM_MAP は、インスタンスまたはエンティティに設定します。
- ・ ブロック RAM が XST で推論できない場合、ロジックが [グローバルな最適化目標 \(-glob_opt\)](#) に渡され、最適化されます。
- ・ このマクロは、推論されません。XST によりロジックがマップされていることを確認してください。

制約値

- ・ yes [または true (XCF)]
- ・ no [または false (XCF)] (デフォルト)

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「[はじめに](#)」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute bram_map: string;
```

次のように指定します。

```
attribute bram_map of component_name: component is "{yes|no}";
```

Verilog の構文例

次をモジュール宣言またはインスタンス化の直前に入力します。

```
(* bram_map = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" bram_map = {yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
  
INST "instance_name" bram_map = {yes|no|true|false};  
  
END;
```

最大ファンアウト

MAX_FANOUT 制約を使用すると、ネットまたは信号のファンアウト数を制限できます。

- ・ 最大ファンアウト制約には、次のような特徴があります。
 - グローバルにもローカルにも設定できます。
 - デフォルト値は 10 万です。値は整数にします。
- ・ ファンアウトが大きいと配線で問題を生じることがあります。このため、XST ではゲートを複製したり、バッファを挿入することでファンアウト数が制限されます。
 - これは技術面での限界ではなく、XST の基準です。
 - この制限は、特に小さい値 (30 未満) に設定されている場合などは、適用されないこともあります。
- ・ ほとんどの場合、ファンアウトが大きいネットを駆動するゲートを複製することでファンアウト数が制限されます。
 - XST でゲートが複製できない場合は、バッファが挿入されます。
 - これらのバッファがインプリメンテーション レベルでロジック トリミングされないようにするには、NGC ファイルで **KEEP** 制約を設定します。
- ・ レジスタの複製オプションを no に設定している場合は、バッファのみを使用してフリップフロップおよびラッチのファンアウト数が制限されます。
- ・ MAX_FANOUT はグローバルに設定できますが、エンティティやモジュール、または信号に個別に設定することも可能です。

最大ファンアウト値よりも実際のネット ファンアウトが小さい場合

- ・ 実際のネット ファンアウトが MAX_FANOUT 値よりも小さい場合は、MAX_FANOUT の設定方法によって XST のビヘイビアーが異なります。
- ・ XST は、次の条件で使用される場合、最大ファンアウト値をガイダンスとしてのみ解釈します。
 - ISE® Design Suite で設定
 - コマンド ラインで設定
 - 特定の階層ブロックに適用
- ・ MAX_FANOUT を特定のネットに設定した場合は、ロジックは複製されません。ネットに設定した場合は、XST で最適なタイミング最適化が行われないことがあります。

たとえば、実際のファンアウトが 80 で、MAX_FANOUT 値が 100 に設定されたネットをクリティカル パスが通過しているとします。

 - 最大ファンアウトが ISE Design Suite で指定される場合、XST はそれを複製して、タイミングを改善しようとします。
 - MAX_FANOUT をネットに設定している場合は、ロジックは複製されません。

reduce 値の最大ファンアウト

- ・ MAX_FANOUT には、reduce という値も設定できます。
- ・ reduce 値には、次のような特徴があります。
 - XST には直接影響しません。
 - 配置配線で適用されます。それまで、ファンアウトの制御は延期されます。
- ・ reduce 値の最大ファンアウトの適用ルールは次のとおりです。
 - ネットにのみ適用できます。
 - グローバルには適用できません。
- ・ XST は指定したネットに関連するロジック最適化をすべてディスエーブルにします。
 - 指定したネットは、合成後のネットリストで保持されます。
 - MAX_FANOUT=reduce プロパティが指定されたネットには適用されています。
- ・ さらに多くのグローバルな MAX_FANOUT 制約を integer 値で定義するには、次の方法を使用します。
 - コマンド ラインを使用して設定
 - ネットを含むエンティティまたはモジュールに属性を付ける
- ・ このようにグローバルな MAX_FANOUT 制約が定義されると、次が実行されます。
 - **reduce** 値が優先されます。
 - そのネットに対する **integer** 値は無視されます。

適用可能エレメント

グローバルに適用するか、またはエンティティ、モジュール、信号に適用できます。

例外 : MAX_FANOUT に **reduce** 値が使用される場合、この制約は信号にのみ適用できます。

適用ルール

設定したエンティティ、モジュール、または信号に適用されます。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute max_fanout: string;
```

次のように指定します。

```
attribute max_fanout of {signal_name|entity_name}:{signal|entity}  
is "integer";
```

または

```
attribute max_fanout of {signal_name}:{signal} is "reduce";
```

Verilog の構文例

次をモジュールまたは信号宣言の直前に入力します。

```
(* max_fanout = "integer" *)
```

または

```
(* max_fanout = "reduce" *)
```

XCF の構文例 1

```
MODEL "entity_name" max_fanout=integer;
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" max_fanout=integer;  
END;
```

XCF の構文例 3

```
BEGIN MODEL "entity_name"  
NET "signal_name" max_fanout="reduce";  
END;
```

XST コマンド ライン構文例

```
-max_fanout integer
```

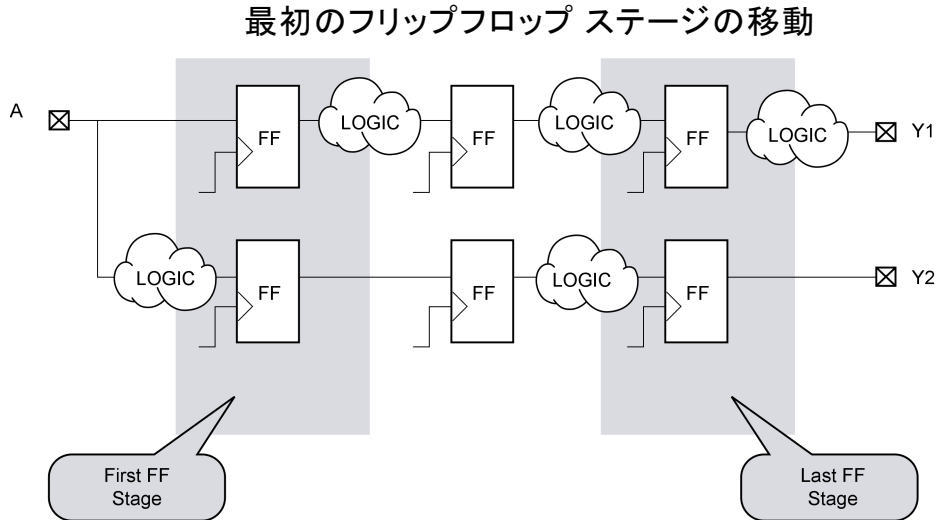
ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Xilinx-Specific Options] → [Max Fanout]

最初のステージの移動

最初のステージの移動 (MOVE_FIRST_STAGE) 制約は、プライマリ入力からのレジスタのリタイミングを制御します。



- ・ フリップフロップは、そのパスがプライマリ入力からの場合は、最初のステージに含まれます。
- ・ フリップフロップのパスがプライマリ出力に向かう場合は、最後のフリップフロップ ステージに含まれます。

レジスタ自動調整

- ・ MOVE_FIRST_STAGE と [MOVE_LAST_STAGE](#) は、[レジスタの自動調整 \(REGISTER_BALANCING\)](#) に関連しています。
- ・ レジスタ バランス (自動調整) 中、フリップフロップは次のように移動されます。
 - 最初の段階にあるフリップフロップは順方向
 - 最終の段階にあるフリップフロップは逆方向
- ・ このプロセスにより、input-to-clock および clock-to-output のタイミングが極度に増加する場合があります。これを避けるには、次を使用します。
 - OFFSET_IN_BEFORE
 - OFFSET_IN_AFTER
- ・ 複数の制約を付けると、[レジスタの自動調整](#) プロセスに影響があります。

その他の制約

- ・ 次の条件の場合、さらに 2 つの制約を使用できます。
 - デザインに必須の要件がない場合、または
 - 最初および最終の段階を変更せずに、最初の結果だけを確認する場合
- ・ 追加できる制約は、次のとおりです。
 - MOVE_FIRST_STAGE
 - MOVE_LAST_STAGE
- ・ どちらの制約も、yes または no に設定できます。
 - MOVE_FIRST_STAGE を no に設定すると、最初のフリップフロップ ステージは移動しません。
 - MOVE_LAST_STAGE を no に設定すると、最終のフリップフロップ ステージは移動しません。

適用可能エレメント

次にのみ適用されます。

- ・ デザイン全体
- ・ 単一のモジュールまたはエンティティ
- ・ プライマリ クロック信号

適用ルール

上の図を参照してください。

制約値

- ・ yes [または true (XCF)] (デフォルト)
- ・ no [または false (XCF)]

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute move_first_stage : string;
```

次のように指定します。

```
attribute move_first_stage of  
{entity_name|signal_name}:{signal|entity} is "{yes|no}";
```

Verilog の構文例

次をモジュールまたは信号宣言の直前に入力します。

```
(* move_first_stage = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" move_first_stage={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "primary_clock_signal" move_first_stage={yes|no|true|false};  
END;
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-move_first_stage {yes|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Xilinx Specific Options] → [Move First Flip-Flop Stage]

最後のステージの移動

最後のステージの移動 (MOVE_LAST_STAGE) 制約では、プライマリ出力にあるレジスタのリタイミングを制御します。

MOVE_LAST_STAGE と MOVE_FIRST_STAGE は、レジスタの自動調整 (REGISTER_BALANCING) に関連しています。

適用可能エレメント

次にのみ適用されます。

- ・ デザイン全体
- ・ 単一のモジュールまたはエンティティ
- ・ プライマリ クロック信号

適用ルール

詳細は、「最初のステージの移動」を参照してください。

制約値

- ・ yes [または true (XCF)] (デフォルト)
- ・ no [または false (XCF)]

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute move_last_stage : string;
```

次のように指定します。

```
attribute move_last_stage of  
{entity_name|signal_name}:{signal|entity} is "{yes|no}";
```

Verilog の構文例

次をモジュールまたは信号宣言の直前に入力します。

```
(* move_last_stage = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" {move_last_stage={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"
```

```
NET "primary_clock_signal" move_last_stage={yes|no|true|false};
```

END ;

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-move_last_stage {yes|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Xilinx Specific Options] → [Move Last Flip-Flop Stage]

乗算器スタイル

乗算器スタイル (MULT_STYLE) 制約を使用すると、マクロ生成プログラムで乗算器マクロをインプリメントする方法を設定できます。

適用可能エレメント

グローバルに適用するか、またはエンティティ、モジュール、信号に適用できます。

適用ルール

- ・ 設定したエンティティ、モジュール、または信号に適用されます。
- ・ これは、HDL 属性からしか適用できず、
- ・ コマンドライン オプションはありません。

制約値

- ・ auto (デフォルト)
各マクロに対して最適なインプリメント方法が自動設定されます。
- ・ block
- ・ pipe_block
DSP48 ベースの乗算器をパイプライン化するために使用します。
- ・ kcm
- ・ csd
- ・ lut
- ・ pipe_lut
スライス ベースの乗算器に使用します。このインプリメンテーション スタイルは、ブロック乗算器や LUT リソースを使用するように手動で指定することもできます。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute mult_style: string;
```

次のように指定します。

```
attribute mult_style of {signal_name|entity_name}:{signal|entity}  
is "{auto|block|pipe_block|kcm|csd|lut|pipe_lut}";
```

Verilog の構文例

次をモジュールまたは信号宣言の直前に入力します。

```
(* mult_style = "{auto|block|pipe_block|kcm|csd|lut|pipe_lut}" *)
```

XCF の構文例 1

```
MODEL "entity_name"  
mult_style={auto|block|pipe_block|kcm|csd|lut|pipe_lut};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
  
NET "signal_name"  
mult_style={auto|block|pipe_block|kcm|csd|lut|pipe_lut};  
  
END;
```

グローバル クロック バッファの数

グローバル クロック バッファ数 (-bufg) コマンド ライン オプションでは、XST で作成される BUFG エLEMENTの最大数を指定できます。

デバイスで使用可能な BUFG 数を超える値は設定できません。

適用可能エレメント

グローバルに適用します。

適用ルール

ありません。

制約値

- ・ 値は整数にします。
- ・ デフォルト値は、次のとおりです。
 - ターゲット デバイスによって異なる
 - 使用可能な BUFG エLEMENTの最大数と同じ

グローバル クロック バッファ数のデフォルト

デバイス	デフォルト値
Spartan®-6	16
Virtex®-6	32
7 series	TBI

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-bufg integer
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Xilinx-Specific Options] → [Number of Clock Buffers]

インスタンス化されたプリミティブの最適化

インスタンス化されたプリミティブの最適化 (OPTIMIZE_PRIMITIVES) を使用すると、インスタンス化済みザイリンクス プリミティブが XST で最適化されない場合、そのデフォルトをディアクティベートします。デフォルトをディアクティベートすると、XST でこれらのプリミティブが最適化されるようになります。

インスタンス化されたプリミティブの最適化の制限

- ・ インスタンス化したプリミティブに **RLOC** のような特定の制約が付いていると、XST でそのまま保持されます。
- ・ XST では、すべてのプリミティブが最適化されるわけではありません。MULT18x18、ブロック RAM、DSP48 などのハードウェア エLEMENT は、インスタンス化したプリミティブの最適化がオンになっていても、最適化 (変更) されません。

適用可能エレメント

階層ブロック、コンポーネント、およびインスタンスに対してグローバルに適用されます。

適用ルール

設定したコンポーネントまたはインスタンスに適用されます。

制約値

- ・ yes [または true (XCF)]
- ・ no [または false (XCF)] (デフォルト)

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

回路図の例

- ・ 有効なインスタンスに設定します。
- ・ 属性名

OPTIMIZE_PRIMITIVES

VHDL の構文例

次のように宣言します。

```
attribute optimize_primitives: string;
```

次のように指定します。

```
attribute optimize_primitives of  
{component_name|entity_name|label_name}:{component|entity|label}  
is "{yes|no}";
```

Verilog の構文例

インスタンス、モジュール、または信号宣言の直前に入力します。

```
(* optimize_primitives = "{yes|no}" *)
```

XCF の構文例

```
MODEL "entity_name" optimize_primitives = {yes|no|true|false};
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-optimize_primitives {yes|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Specific Options] → [Optimize Instantiated Primitives]

I/O レジスタの IOB 内へのパック

入力/出力のパス タイミングを向上するため、フリップフロップを I/O 内に配置します。

auto に設定されると、最適化設定によって XST で実行される内容は異なります。

- ・ area

デザインに占めるスライス数を削減するために、レジスタはできるだけ多く IOB に含まれます。

- ・ speed

XST では、タイミング制約でカバーされない（タイミングの最適化が考慮されない）IOB にレジスタがパックされます。

- ◆ たとえば、**PERIOD** 制約を指定していて、XST で PERIOD 制約でカバーされないと判断されると、レジスタがその IOB にパックされます。
- ◆ このようなタイミング最適化制約でカバーされるレジスタを IOB に含める場合は、このレジスタに IOB 制約を個別に設定する必要があります。

この制約の詳細は、『[制約ガイド](#)』（UG625）を参照してください。

電力削減

電力削減 (POWER) 制約を使用すると、合成最適化手法を使用して、消費電力を削減できます。

- ・ この最適化は、デフォルトではオフになっています。
- ・ この制約がイネーブルになっていても、XST では**最適化目標 (OPT_MODE)** で指定された主な最適化目標 (エリアまたは速度) を優先します。
- ・ 消費電力を削減する最適化を実行することで、主な最適化目標に悪影響を及ぼさないかどうか、注意してください。
- ・ 電力最適化は、主にブロック RAM に関連しています。XST では、RAM イネーブル機能を使用することで、同時にアクティブになるブロック RAM の数を最小限に抑えようとしています。

RAM の電力最適化の詳細は、「[RAM スタイル](#)」を参照してください。

適用可能エレメント

次に適用します。

- ・ component または entity (VHDL)
- ・ model または label (instance) (Verilog)
- ・ model または INST (model 内) (XCF)
- ・ デザイン全体 (XST コマンド ライン)

適用ルール

設定したエンティティ、モジュール、または信号に適用されます。

制約値

- ・ yes [または true (XCF)]
- ・ no [または false (XCF)] (デフォルト)

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「[はじめに](#)」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute power: string;
```

次のように指定します。

```
attribute power of {component_name|entity_name}  
: {component|entity} is "{yes|no}";
```

Verilog の構文例

次をモジュール宣言またはインスタンス化の直前に入力します。

```
(* power = "{yes|no}" *)
```

XCF の構文例

```
MODEL "entity_name" power = {yes|no|true|false};
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-power {yes|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Power Reduction]

RAM の抽出

RAM の抽出 (RAM_EXTRACT) 制約を使用すると、RAM マクロの推論を有効または無効にできます。

適用可能エレメント

グローバルに適用するか、またはエンティティ、モジュール、信号に適用できます。

適用ルール

設定したエンティティ、モジュール、または信号に適用されます。

制約値

- ・ yes [または true (XCF)] (デフォルト)
- ・ no [または false (XCF)]

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute ram_extract: string;
```

次のように指定します。

```
attribute ram_extract of  
{signal_name|entity_name}:{signal|entity} is "{yes|no}";
```

Verilog の構文例

次をモジュールまたは信号宣言の直前に入力します。

```
(* ram_extract = "{yes|no}" *)
```

XCF の構文例 1

```
RAM Extraction Syntax MODEL "entity_name"  
ram_extract={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" ram_extract={yes|no|true|false};  
END;
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-ram_extract {yes|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [RAM Extraction]

RAM スタイル

RAM スタイル (RAM_STYLE) 制約では、推論された RAM マクロのインプリメント方法を指定できます。

block_power1 および block_power2 を使用すると、ブロックリソースへインプリメントされる RAM の消費電力の削減を目的にした 2 レベルの最適化が可能になります。

block_power1

- ・ **最適化目標 (OPT_MODE)** で指定された主な最適化目標 (エリアまたは速度) への影響を最小限に抑えます。
- ・ 一般的な電力最適化が **電力削減 (POWER)** 制約でイネーブルになっている場合に選択されるモードです。
- ・ 次にのみ指定できます。
 - VHDL 属性
 - Verilog 属性
 - XCF 制約

block_power2

- ・ さらなる電力削減が可能
- ・ エリアおよび速度に多大な影響をもたらす可能性あり
- ・ 次にのみ指定できます。
 - VHDL 属性
 - Verilog 属性
 - XCF 制約

これらの最適化方法の詳細は、「**ブロック RAM の電力削減**」を参照してください。

適用可能エレメント

グローバルに適用するか、またはエンティティ、モジュール、信号に適用できます。

適用ルール

設定したエンティティ、モジュール、または信号に適用されます。

制約値

- ・ auto (デフォルト)
次に基づいて、推論された各 RAM に対する最適なインプリメント方法が XST で設定されます。
 - ブロック RAM のインプリメンテーション (同期データ読み出し) が可能な記述スタイルかどうか
 - 使用可能なブロック RAM リソース
- ・ distributed
分散 RAM リソースにインプリメンテーションされます。
- ・ pipe_distributed
 - 推論された RAM が LUT リソースにインプリメントされ、そのサイズに合わせるために複数の分散 RAM プリミティブが必要な場合、RAM のデータ出力パスにマルチプレクサ ロジックが作成されます。XST は pipe_distributed の値により、使用可能なレイテンシ ステージをすべて使用し、このロジックをパイプライン処理します。
 - 次にのみ指定できます。
 - ◆ VHDL 属性
 - ◆ Verilog 属性
 - ◆ XCF 制約
- ・ block
ブロック RAM リソースにインプリメンテーションされます。ブロック RAM への実際のインプリメンテーションは、次の条件によって決まります。
 - 正しく同期されたデータ読み出し、および
 - デバイスで使用可能なリソース

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute ram_style: string;
```

次のように指定します。

```
attribute ram_style of {signal_name|entity_name}: {signal|entity} is  
"{auto|block|distributed|pipe_distributed|block_power1|block_power2}";
```

Verilog の構文例

次をモジュールまたは信号宣言の直前に入力します。

```
(* ram_style = "{auto|block|distributed|pipe_distributed|block_power1|block_power2}" *)
```

XCF の構文例 1

```
MODEL "entity_name" ram_style={auto|block|distributed|pipe_distributed|block_power1|block_power2};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
    NET "signal_name" ram_style={auto|block|distributed|pipe_distributed|block_power1|block_power2};  
END;
```

XST コマンド ライン 構文例

run コマンドでグローバルに設定します。

-ram_style {auto|block|distributed}

コマンド ラインからは、pipe_distributed には設定できません。

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [RAM Style]

コアの読み込み

コアの読み込み (READ_CORES) 制約を使用すると、タイミングを概算したり、デバイスの使用率を指定するために、EDIF または NGC コア ファイルを XST に読み込むかどうかを指定できます。

- ・ 特定のコアを読み込むとロジックの接続方法が認識されるので、XST でそのコアの周囲のロジックの最適化が改善されます。
- ・ コア別にコアを読み込むかどうかを指定できます。
- ・ この制約はデisableにする必要のあることがあります。たとえば、PCI™ コアに直接接続されるロジックはほかのコアとは異なる方法で最適化されるため、は XST に認識されないようにする必要があります。

詳細については、「[コアの処理](#)」を参照してください。

適用可能エレメント

次に適用します。

- ・ component または entity (VHDL)
- ・ model または label (instance) (Verilog)
- ・ model または INST (model 内) (XCF)
- ・ デザイン全体 (XST コマンド ライン)

次の規則が適用されます。

- ・ この制約は [ボックス タイプ \(BOX_TYPE\)](#) と一緒に使用できるので、制約が適用されるオブジェクト セットは同じである必要があります。
- ・ READ_CORES が少なくとも 1 ブロックの単一インスタンスに適用される場合は、このブロックのほかのインスタンスすべてにも適用されます。

適用ルール

ありません。

制約値

- ・ yes [または true (XCF)] (デフォルト)
コアはプロセスされますが、ブラックボックスとして維持され、デザインに組み込まれません。
- ・ no [または false (XCF)]
コアはプロセスされません。
- ・ **optimize**
コアはプロセスされ、コアのネットリストがデザイン全体にマージされます。この値は、[コマンド ライン モード](#)でのみ使用できます。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「[はじめに](#)」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute read_cores: string;
```

次のように指定します。

```
attribute read_cores of {component_name|entity_name}  
: {yes|no|optimize}"; component|entity is "{yes|no|optimize}";
```

Verilog の構文例

次をモジュール宣言またはインスタンス化の直前に入力します。

```
(* read_cores = "{yes|no|optimize}" *)
```

XCF の構文例 1

```
MODEL "entity_name" read_cores = {yes|no|true|false|optimize};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
INST "instance_name" read_cores = {yes|no|true|false|optimize};  
END;
```

XST コマンド ライン構文例

```
-read_cores {yes|no|optimize}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

Process > Properties > Synthesis Options > Read Cores

ISE® Design Suite からは、optimize オプションは指定できません。

制御セットの削減

制御セットの削減 (REDUCE_CONTROL_SETS) 制約を使用すると、制御セットの数を削減できます。

- ・ 制御セットの数を削減すると、次が実現されるはずです。
 - デザイン エリアの削減
 - マップのパッキング プロセスの改善
 - LUT 数が増加してもスライス数を削減
- ・ 制御セットの削減には、次のような特徴があります。
 - 同期制御信号にのみ適用します。
 - ◆ 同期セット/リセット
 - ◆ クロック イネーブル
 - 非同期セット/リセット ロジックには使用しても何の効果もありません。

適用可能エレメント

グローバルに適用します。

適用ルール

ありません。

制約値

- ・ auto (デフォルト)
制御セットの最適化が実行されます。
- ・ no
制御セットの最適化は実行されません。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-reduce_control_sets {auto|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Reduce Control Sets]

レジスタ自動調整

レジスタ自動調整 (REGISTER_BALANCING) 制約を使用すると、フリップフロップのリタイミングを有効にできます。

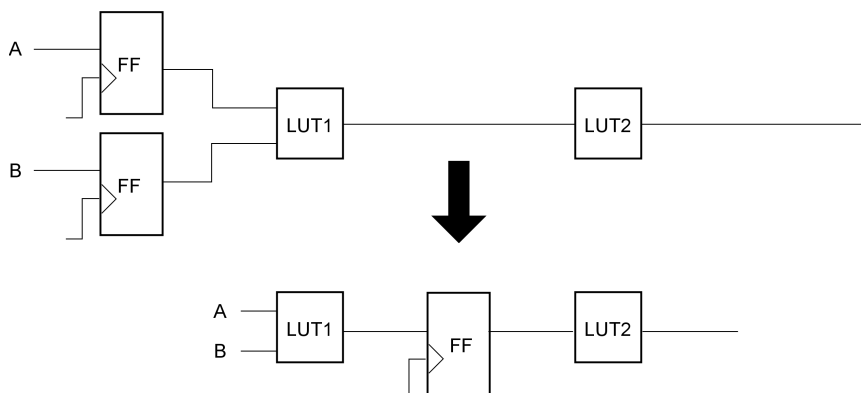
- ・ レジスタ自動調整とは、クロック周波数を増加するためにフリップフロップおよびラッチの位置を変更する手法です。
- ・ この制約が有効になっていると、XST で別のクロックドメインの境界へ組み合わせロジックが移動できるようになります。クロックドメイン間でロジックが移動されないようにするには、次のいずれかを実行します。
 - [Register Balancing] の値を NO に設定してレジスタ自動調整のオプションをオフにします。
 - XCF のクロス クロックドメインに False パス (TIG) 制約を指定します。
 - 信号に KEEP 制約を指定すると、それらの信号をクロスするフリップフロップは使用できません。
- ・ レジスタ自動調整には、次の 2 種類があります。
 - 順方向のレジスタ自動調整
 - 逆方向のレジスタ自動調整

順方向のレジスタ自動調整

- ・ LUT の各入力にあるフリップフロップすべてを 1 つのフリップフロップとして LUT の出力に移動します。
- ・ 複数のフリップフロップが 1 つのフリップフロップに置き換わる際には、次のように LUT 名に基づいた名前が選択されます。

`LutName_FRBId`

順方向のレジスタ自動調整

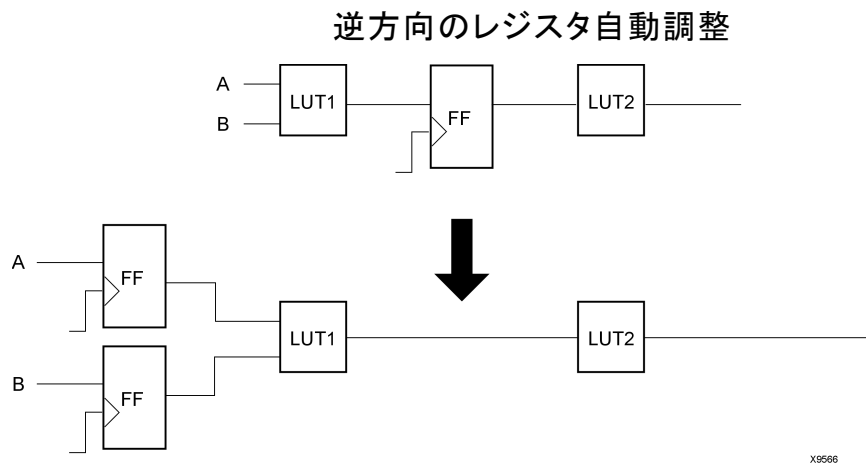


X9563

逆方向のレジスタ自動調整

- ・ LUT の出力にある 1 つのフリップフロップを LUT のフリップフロップの各入力に移動します。
- ・ フリップフロップ数は増減する可能性があります。
- ・ 新しいフリップフロップには、オリジナルのフリップフロップ名の後に接尾辞が付きます。

*OriginalFFName_***BRB***Id*

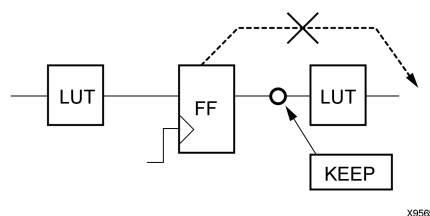


レジスタ自動調整に影響するその他の制約

- ・ 次の制約もレジスタ自動調整に影響を与えます。
 - 最初のフリップフロップ ステージの移動
 - 最後のフリップフロップ ステージの移動
- ・ 次の制約もレジスタ自動調整に影響を与えます。
 - 階層保持 (KEEP_HIERARCHY)
 - ◆ 階層を保持している場合、フリップフロップはブロックの境界内でのみ移動します。
 - ◆ 階層をフラットにした場合、フリップフロップはブロックの境界外にも移動します。
 - I/O レジスタの IOB 内へのパック (IOB)

IOB=TRUE の場合、設定したフリップフロップにレジスタ自動調整は適用されません。
 - インスタンス化されたプリミティブの最適化 (OPTIMIZE_PRIMITIVES)
 - ◆ インスタンス化されたフリップフロップは、OPTIMIZE_PRIMITIVES=YES の場合にのみ移動されます。
 - ◆ フリップフロップは、OPTIMIZE_PRIMITIVES=YES の場合にのみインスタンス化されたプリミティブ間で移動されます。
 - キープ (KEEP)
 - ◆ この制約を出力フリップフロップ信号に適用した場合、フリップフロップは順方向には移動できません。次の図を参照してください。
 - ◆ 出力フリップフロップ信号に適用した場合、フリップフロップは逆方向には移動できません。
 - ◆ フリップフロップの入力と出力の両方に適用すると、REGISTER_BALANCE=NO と同じになります。

出力フリップフロップに適用した場合



適用可能エレメント

レジスタ自動調整制約の適用

- コマンド ラインまたは ISE® Design Suite を使用してデザイン全体にグローバルに適用
- エンティティまたはモジュールに対して適用
- フリップフロップ記述 (RTL) に対応する信号に対して適用
- フリップフロップ インスタンスに対して適用
- プライマリ クロック信号に対して適用

適用ルール

設定したエンティティ、モジュール、または信号に適用されます。

制約値

- ・ yes [または true (XCF)]
順方向および逆方向どちらのリタイミングも可能になります。
- ・ no [または false (XCF)] (デフォルト)
フリップフロップのリタイミングはどちらの方向も実行されません。
- ・ forward
順方向のリタイミングのみができます。
- ・ backward
逆方向のリタイミングのみができます。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。
詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute register_balancing: string;
```

次のように指定します。

```
attribute register_balancing of  
{signal_name|entity_name}:{signal|entity}  
is "{yes|no|forward|backward}";
```

Verilog の構文例

次をモジュールまたは信号宣言の直前に入力します。

```
* register_balancing = "{yes|no|forward|backward}" *) (
```

XCF の構文例 1

```
MODEL "entity_name"  
  
register_balancing={yes|no|true|false|forward|backward};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"

NET "primary_clock_signal"
register_balancing={yes|no|true|false|forward|backward};

END;
```

XCF の構文例 3

```
BEGIN MODEL "entity_name"

INST "instance_name"

register_balancing={yes|no|true|false|forward|backward};

END;
```

XST コマンド ラインの構文例

run コマンドでグローバルに設定します。

```
-register_balancing {yes|no|forward|backward}
```

ISE Design Suite での設定

ISE Design Suite で次のようにグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Xilinx-Specific Options] → [Register Balancing]

レジスタの複製

レジスタの複製 (REGISTER_DUPLICATION) 制約を使用すると、レジスタの複製を有効または無効にできます。

レジスタの複製は、次の理由から実行されます。

- ・ タイミング最適化の一部として実行され、ファンアウトの多いレジスタが複製されます。
- ・ IOB 制約では、タイミング制約の付いたレジスタに IOB=true 属性が付いている場合複製されます。
- ・ **MAX_FANOUT** 制約が適用される場合、レジスタのファンアウトが適用した max_fanout 値を超えると複製されます。

適用可能エレメント

グローバルに適用するか、またはエンティティ、モジュール、信号に適用できます。

適用ルール

設定したエンティティまたはモジュールに適用されます。

制約値

- ・ yes [または true (XCF)] (デフォルト)
- ・ no [または false (XCF)]

REGISTER_DUPLICATION が yes に設定されると、レジスタ複製が次のようになります。

- ・ 有効になります。
- ・ タイミング最適化およびファンアウト制御の段階で実行されます。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute register_duplication: string;
```

エンティティに対して次のように指定します。

```
attribute register_duplication of entity_name: entity is  
"{yes|no}";
```

信号に対して次のように指定します。

```
attribute register_duplication of signal_name: signal is  
"{yes|no}";
```

Verilog の構文例

次をモジュール宣言、インスタンスエーションまたは信号宣言の直前に入力します。

```
(* register_duplication = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" register_duplication={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" register_duplication={yes|no|true|false};  
END;
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-register_duplication {yes|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Xilinx-Specific Options] → [Register Duplication]

ROM の抽出

ROM_EXTRACT を使用すると、ROM マクロの推論を有効にできます。ROM は通常、割り当てられた値がすべて定数である case 文から推論されます。

適用可能エレメント

グローバルに適用するか、デザイン エレメントまたは信号に適用します。

適用ルール

設定したエンティティ、モジュール、または信号に適用されます。

制約値

- ・ yes [または true (XCF)] (デフォルト)
- ・ no [または false (XCF)]

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute rom_extract: string;
```

次のように指定します。

```
attribute rom_extract of  
{signal_name|entity_name}:{signal|entity} is "{yes|no}";
```

Verilog の構文例

次をモジュールまたは信号宣言の直前に入力します。

```
(* rom_extract = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" rom_extract={yes|no|true|false};*
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" rom_extract={yes|no|true|false};  
END;
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-rom_extract {yes|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [ROM Extraction]

ROM スタイル

ROM スタイル (ROM_STYLE) 制約では、推論された ROM マクロのインプリメント方法を指定します。

- ・ ROM_STYLE を使用する場合、まず [ROM の抽出 \(ROM_EXTRACT\)](#) を yes に設定しておく必要があります。
- ・ 推論された各 ROM に対して最適なインプリメント方法が自動設定されます。
- ・ このインプリメンテーション スタイルは、ブロック RAM や LUT リソースを使用するように手動で指定することもできます。

適用可能エレメント

グローバルに適用するか、またはエンティティ、モジュール、信号に適用できます。

適用ルール

設定したエンティティ、モジュール、または信号に適用されます。

制約値

- ・ auto (デフォルト)
- ・ block
- ・ distributed

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

ROM_STYLE を使用する場合、まず [ROM の抽出 \(ROM_EXTRACT\)](#) を yes に設定しておく必要があります。

次のように宣言します。

```
attribute rom_style: string;
```

次のように指定します。

```
attribute rom_style of {signal_name|entity_name}:{signal|entity}  
is "{auto|block|distributed}";
```

Verilog の構文例

次をモジュールまたは信号宣言の直前に入力します。

```
(* rom_style = "{auto|block|distributed}" *)
```

XCF の構文例 1

ROM_STYLE を使用する場合、まず [ROM の抽出 \(ROM_EXTRACT\)](#) を yes に設定しておく必要があります。

```
MODEL "entity_name" rom_style={auto|block|distributed};
```

XCF の構文例 2

ROM_STYLE を使用する場合、まず [ROM の抽出 \(ROM_EXTRACT\)](#) を yes に設定しておく必要があります。

```
BEGIN MODEL "entity_name"  
  
NET "signal_name" rom_style={auto|block|distributed};  
  
END;
```

XST コマンド ライン構文例

ROM_STYLE を使用する場合、まず [ROM の抽出 \(ROM_EXTRACT\)](#) を yes に設定しておく必要があります。

run コマンドでグローバルに設定します。

```
-rom_style {auto|block|distributed}
```

ISE Design Suite での設定

ROM_STYLE を使用する場合、まず [ROM の抽出 \(ROM_EXTRACT\)](#) を yes に設定しておく必要があります。

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [ROM Style]

シフトレジスタの抽出

シフトレジスタの抽出 (SHREG_EXTRACT) 制約では、シフトレジスタ マクロの推論を有効にできます。

- ・ オンにすると、SRL16 および SRLC16 のような専用ハードウェアリソースが使用されます。
- ・ 詳細は、第 7 章「HDL コーディング手法」を参照してください。

適用可能エレメント

グローバルに適用するか、デザイン エレメントまたは信号に適用します。

適用ルール

設定したデザイン エレメントまたは信号に適用されます。

制約値

- ・ yes [または true (XCF)] (デフォルト)
- ・ no [または false (XCF)]

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute shreg_extract : string;
```

次のように指定します。

```
attribute shreg_extract of  
{signal_name|entity_name}:{signal|entity} is "{yes|no}";
```

Verilog の構文例

次をモジュールまたは信号宣言の直前に入力します。

```
(* shreg_extract = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" shreg_extract={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" shreg_extract={yes|no|true|false};  
END;
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-shreg_extract {yes|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [Shift Register Extraction]

シフトレジスタの最小サイズ

シフトレジスタの最小サイズ (SHREG_MIN_SIZE) 制約を使用すると、SRL タイプのリソースを使用して推論およびインプリメントされるシフトレジスタの最小の長さを制御できます。

- ・ 指定した制限を下回るシフトレジスタは、単純なフリップフロップを使用してインプリメントされます。
- ・ 2 ビットのシフトレジスタのような小型のシフトレジスタ マクロをインプリメントするために SRL タイプのリソースを使用しすぎると、ほかの要素に対する配置制限が増え、回路パフォーマンスに悪影響の出ることもあります。
- ・ SHREG_MIN_SIZE を使用すると、XST で単純なフリップフロップ リソースを使用して指定する長さを下回るシフトレジスタがインプリメントされるようになります。
- ・ Spartan®-6 デバイスの場合、次の 4 スライスすべてに対して 1 つの SliceM が使用できます。
 - SliceL
 - SliceM
 - SliceX
 - SliceY

このため、エレメントが特に貴重になり、LUT RAM アプリケーションなどに使用するように節約される可能性があります。

適用可能エレメント

- ・ この制約は、XST オプションとしてのみ適用でき、デザイン全体のグローバル推論のしきい値を定義します。
- ・ 各シフトレジスタの推論をさらに詳細に制御する必要がある場合は、この制約を [SHREG_EXTRACT](#) 制約と併用します。SHREG_EXTRACT は該当するデザイン エレメントに適用できます。

適用ルール

ありません。

制約値

値は整数にします。

- ・ 2 またはそれ以上の自然値を使用します。
- ・ デフォルトの値は 2 です。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-shreg_min_size integer
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [Shift Register Minimum Size]

スライス (LUT-FF ペア) 使用率

スライス (LUT-FF ペア) 使用率 (SLICE_UTILIZATION_RATIO) 制約を使用すると、タイミング最適化における LUT-FF ペアのエリア サイズの上限を指定できます。

- ・ LUT-FF ペアのエリア サイズは、絶対値またはパーセントで定義します。
- ・ このエリア制約を満たすことができない場合は、エリア制約を無視してタイミング最適化が実行されます。
- ・ 自動的にリソースが管理されないようにするには、-1 を指定します。

詳細は、「[エリア制約を設定した場合のスピード最適化](#)」を参照してください。

適用可能エレメント

グローバルに、または VHDL エンティティまたは Verilog モジュールに適用します。

適用ルール

設定したエンティティまたはモジュールに適用されます。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「[はじめに](#)」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute slice_utilization_ratio: string;
```

次のように指定します。

```
attribute slice_utilization_ratio of entity_name : entity is  
"integer";
```

```
attribute slice_utilization_ratio of entity_name : entity is  
"integer%";
```

```
attribute slice_utilization_ratio of entity_name : entity is  
"integer#";
```

これらの例の場合、XST では次が実行されます。

- ・ 整数値は最初の 2 つの例ではパーセントとして処理されます。
- ・ 最後の例ではスライスまたは FF/LUT ペアの絶対数として処理されます。

Verilog の構文例

次をモジュール宣言またはインスタンス化の直前に入力します。

```
(* slice_utilization_ratio = "integer" *)
```

```
(* slice_utilization_ratio = "integer%" *)
```

```
(* slice_utilization_ratio = "integer#" *)
```

これらの例の場合、XST では次が実行されます。

- ・ 整数値は最初の 2 つの例ではパーセントとして処理されます。
- ・ 最後の例ではスライスまたは FF/LUT ペアの絶対数として処理されます。

XCF の構文例 3

```
MODEL "entity_name" slice_utilization_ratio="integer#";*
```

- ・ この例では、次の操作が実行されます。
 - 整数値は最初の 2 行ではパーセントとして処理されます。
 - 最後の行ではスライスまたは FF/LUT ペアの絶対数として処理されます。
- ・ 整数値と % および # 文字の間にはスペースを入れないでください。
- ・ % が使用されるか、% と # のどちらも使用されない場合、整数値の範囲は -1 ~ 100 です。
- ・ 整数値と % または # 文字を二重引用符 (" ") で囲んでください。% および # は XCF (ザイリンクス制約ファイル) の特殊文字です。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-slice_utilization_ratio integer
```

```
-slice_utilization_ratio integer%
```

```
-slice_utilization_ratio integer#
```

これらの例の場合、XST では次が実行されます。

- ・ 整数値は最初の 2 行ではパーセントとして処理されます。
- ・ 最後の行ではスライスまたは FF/LUT ペアの絶対数として処理されます。

% が使用されるか、% と # のどちらも使用されない場合、整数値の範囲は -1 ~ 100 です。

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

- ・ [Process Properties] ダイアログ ボックス → [Synthesis Options] → [Slice Utilization Ratio]
- ・ [Process Properties] ダイアログ ボックス → [Synthesis Options] → [LUT-FF Pairs Utilization Ratio]

ISE® Design Suite の場合 :

- ・ この値は % としてのみ定義できます。
- ・ スライスの絶対値は指定できません。

スライス (LUT-FF ペア) 使用率の許容範囲

スライス (LUT-FF ペア) 使用率の許容範囲 (SLICE_UTILIZATION_RATIO_MAXMARGIN) 制約では、[SLICE_UTILIZATION_RATIO](#) の許容範囲を設定します。

- ・ パラメーターの値は、次のように定義できます。
 - パーセンテージまたは
 - スライスまたは LUT-FF ペアの絶対数
- ・ スライス使用率がこの制約で指定したマージン値の範囲内であれば、制約は満たされていると判断され、タイミング最適化が実行されます。

詳細は、「[エリア制約を設定した場合のスピード最適化](#)」を参照してください。

適用可能エレメント

グローバルに、または VHDL エンティティまたは Verilog モジュールに適用します。

適用ルール

設定したエンティティまたはモジュールに適用されます。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「[はじめに](#)」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute slice_utilization_ratio_maxmargin: string;
```

次のように指定します。

```
attribute slice_utilization_ratio_maxmargin of entity_name :  
entity is "integer";
```

```
attribute slice_utilization_ratio_maxmargin of entity_name :  
entity is "integer%";
```

```
attribute slice_utilization_ratio_maxmargin of entity_name :  
entity is "integer#";
```

- ・ 整数値は最初の 2 つの例ではパーセントとして処理されます。
- ・ 最後の例ではスライスまたは FF/LUT ペアの絶対数として処理されます。
- ・ 次の場合、integer の範囲は 0 ～ 100 です。
 - % が使用される場合、または
 - % と # のどちらも削除される場合

Verilog の構文例

次をモジュール宣言またはインスタンス化の直前に入力します。

```
(* slice_utilization_ratio_maxmargin = "integer" *)  
(* slice_utilization_ratio_maxmargin = "integer%" *)  
(* slice_utilization_ratio_maxmargin = "integer#" *)
```

- ・ 整数値は最初の 2 つの例ではパーセントとして処理されます。
- ・ 最後の例ではスライスまたは FF/LUT ペアの絶対数として処理されます。

XCF の構文例 3

```
MODEL "entity_name" slice_utilization_ratio_maxmargin="integer#";
```

- ・ 整数値は最初の 2 行ではパーセントとして処理されます。
- ・ 最後の行ではスライスまたは FF/LUT ペアの絶対数として処理されます。
- ・ 整数値と % および # 文字の間にはスペースを入れないでください。
- ・ % および # は の特殊文字なので、整数値と % または # 文字を二重引用符 (" ") で囲んでください。
- ・ % が使用されるか、% と # のどちらも使用されない場合、整数値の範囲は 0 ~ 100 です。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-slice_utilization_ratio_maxmargin integer  
-slice_utilization_ratio_maxmargin integer%  
-slice_utilization_ratio_maxmargin integer#
```

上記の例で、整数値は最初の 2 つの例ではパーセントとして処理され、最後の例ではスライスまたは FF/LUT ペアの絶対数として処理されます。

% が使用されるか、% と # のどちらも使用されない場合、整数値の範囲は 0 ~ 100 です。

キャリー チェーンの使用

キャリー チェーンの使用 (USE_CARRY_CHAIN) 制約には、次のような特徴があります。

- ・ グローバルにもローカルにも設定できます。
- ・ マクロ生成時にキャリー チェーンの使用を無効にできます。

XST では、一部のマクロをインプリメントする際にキャリー チェーン リソースが使用されますが、キャリー チェーンを使用しない方が良い結果が得られる場合があります。

適用可能エレメント

グローバルに適用されるか、信号に適用されます。

適用ルール

設定された信号に適用されます。

制約値

- ・ yes [または true (XCF)] (デフォルト)
- ・ no [または false (XCF)]

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

回路図の例

- ・ 有効なインスタンスに設定します。
- ・ 属性名
USE_CARRY_CHAIN

VHDL の構文例

次のように宣言します。

```
attribute use_carry_chain: string;
```

次のように指定します。

```
attribute use_carry_chain of signal_name: signal is "{yes|no}";
```

Verilog の構文例

次を信号宣言の直前に入力します。

```
(* use_carry_chain = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" use_carry_chain={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"
```

```
NET "signal_name" use_carry_chain={yes|no|true|false};  
END;
```

XST コマンド ライン 構文例

run コマンドでグローバルに設定します。

```
-use_carry_chain {yes|no}
```

クロック イネーブルの使用

クロック イネーブル (USE_CLOCK_ENABLE) 制約を使用すると、フリップフロップのクロック イネーブルの使用を有効または無効にできます。

- ・ この制約は通常 ASIC のプロトタイプではオフになります。
- ・ 制約値を no に設定すると、最終インプリメンテーションでクロック イネーブル リソースが使用されません。
- ・ また、デザインによっては、フリップフロップのデータ入力にクロック イネーブルを付けることで、ロジックが最適化され、結果が改善されることもあります。
- ・ auto に設定すると、次が比較検討されます。
 - フリップフロップ入力の専用クロック イネーブル入力を使用した方がいいか
 - フリップフロップの D 入力にクロック イネーブル ロジックを使用した方がいいか
- ・ ユーザーがフリップフロップをインスタンス化すると、XST では [インスタンス化されたプリミティブの最適化 \(OPTIMIZE_PRIMITIVES\)](#) が yes の場合にのみ、クロック イネーブルが削除されます。

適用可能エレメント

次に適用します。

- ・ XST コマンド ラインからデザイン全体に適用
- ・ 特定ブロック (entity、architecture、component)
- ・ フリップフロップを表す信号
- ・ インスタンス化されたフリップフロップを表すインスタンス

適用ルール

設定したエンティティ、コンポーネント、モジュール、信号、インスタンスに適用されます。

制約値

- ・ auto (デフォルト)
- ・ yes [または true (XCF)]
- ・ no [または false (XCF)]

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute use_clock_enable: string;
```

次のように指定します。

```
attribute use_clock_enable of  
{entity_name|component_name|signal_name|instance_name}  
: {entity|component|signal|label} is "{auto|yes|no}";
```

Verilog の構文例

インスタンス、モジュール、または信号宣言の直前に入力します。

```
(* use_clock_enable = "{auto|yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" use_clock_enable={auto|yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" use_clock_enable={auto|yes|no|true|false};  
END;
```

XCF の構文例 3

```
BEGIN MODEL "entity_name"  
INST "instance_name" use_clock_enable={auto|yes|no|true|false};  
END;
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-use_clock_enable {auto|yes|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Xilinx-Specific Options] → [Use Clock Enable]

DSP ブロックの使用

DSP ブロックの使用 (USE_DSP48) 制約を使用すると、DSP ブロックリソースの使用をイネーブルまたはディスエーブルにできます。

DSP ブロック リソース

- ・ **DSP 使用率 (DSP_UTILIZATION_RATIO)** 制約を auto モードおよび automax モードにすると、合成で使用される DSP ブロックリソースの数をさらに制御できます。使用可能な DSP ブロックがすべて使用可能であると認識されます。
- ・ 乗加減算器、乗累算器のようなマクロは次の単純なマクロを統合したものとして扱われます。
 - 乗算器
 - アキュムレータ
 - レジスタ
- ・ XST では、これらの統合を積極的に実行し、パフォーマンスを最大にします。XST は DSP ブロックですべてのパイプライン ステージを使用しようとしています。
- ・ これらの基本的なマクロの DSP ブロックへの統合は、**キープ (KEEP)** 制約を使用して制御します。たとえば、乗算オペランドの前にレジスタ ステージが 2 つ使用可能な場合、それらの間に KEEP 制約を挿入すると、どちらか 1 つが DSP ブロックにインプリメントされないようになります。
- ・ サポートされるマクロおよびインプリメンテーション制御の詳細は、「**HDL コーディング手法**」を参照してください。

適用可能エレメント

- ・ XST コマンドラインからデザイン全体に適用
- ・ 特定ブロック (entity、architecture、component)
- ・ RTL レベルで記述されるマクロを表す信号

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

制約値

- ・ auto (デフォルト)

XST は選択的に演算ロジックを DSP ブロックへインプリメントし、回路パフォーマンスが最高になるようにします。

- 次のようなマクロは、DSP ブロック インプリメンテーションの候補として考慮されます。

- ◆ 乗算
- ◆ 乗加減算
- ◆ 乗累算

- XST は DSP ブロックのカスケード機能をできるだけ使用しようとします。

- 次のようなその他のマクロは、スライス ロジックにインプリメントされます。

- ◆ 加算器
- ◆ カウンタ
- ◆ スタンドアロンのアキュムレータ

- ・ automax

XST は選択したデバイスで使用可能なリソースの制限内で DSP ブロックを最大限に使用します。

- auto モードで考慮されたマクロだけでなく、automax でも次が DSP ブロック インプリメンテーションの候補として考慮されます。

- ◆ 加算器
- ◆ カウンタ
- ◆ スタンドアロンのアキュムレータ

- 主な目的が集積度であり、LUT リソースを空けようとする場合は、automax の使用をお勧めします。

Attention automax を使用すると、デフォルトの auto モードに比べて回路パフォーマンスが落ちます。主なインプリメンテーション目的がパフォーマンスにある場合は、automax を使用しないでください。

- ・ yes [または true (XCF)]

DSP ブロックへの演算ロジックのインプリメンテーションを強制できます。

- 主に yes を使用し、ファンクションを個別に DSP リソースへ強制的にインプリメントします。

- XST ではこのモードで実際に使用可能な DSP リソースが確認されず、DSP ブロックが使用されすぎてしまう可能性があるため、yes の使用はお勧めしません。

Attention yes にして DSP ブロックにファンクションをインプリメントする場合、選択デバイスで実際に使用可能な DSP リソース量と `DSP_UTILIZATION_RATIO` 制約で定義された最大アロケーション (割合) の両方が無視されます。この結果、使用可能な、またはバジェットよりも多く DSP リソースが使用される可能性があります。

- ・ no [または false (XCF)]

DSP ブロックへ該当ロジックがインプリメンテーションされないようにできます。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。
詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute use_dsp48: string;
```

次のように指定します。

```
attribute use_dsp48 of  
"entity_name|component_name|signal_name">{entity|component|signal}  
is "{auto|automax|yes|no}";
```

Verilog の構文例

インスタンス、モジュール、または信号宣言の直前に入力します。

```
(* use_dsp48 = "{auto|automax|yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" use_dsp48={auto|automax|yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" use_dsp48={auto|automax|yes|no|true|false};  
END;
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-use_dsp48 {auto|automax|yes|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [Use DSP Block]

ロー スキュー ラインの使用

ロー スキュー ラインの使用 (USELOWSKEWLINES) 制約には、次の特徴があります。

- ・ 基本的な配線制約です。
- ・ [MAX_FANOUT \(最大ファンアウト数\)](#) 制約の値に基づいて専用クロック リソースおよびロジックの複製が使用されないようにします。
- ・ ネットでロー スキュー 配線リソースを使用するよう指定します。

この制約の詳細は、『[制約ガイド](#)』(UG625)を参照してください。

同期セットの使用

同期セットの使用 (USE_SYNC_SET) 制約を使用すると、フリップフロップの同期セットの使用を有効または無効にできます。

- ・ この制約は通常 ASIC のプロトタイプではオフになります。
- ・ 制約値を no にすると、最終インプリメンテーションで同期セットリソースが使用されません。
- ・ また、デザインによっては、フリップフロップのデータ入力にクロック イネーブルを付けることで、ロジックが最適化され、結果が改善されることもあります。
- ・ auto に設定すると、次が比較検討されます。
 - フリップフロップ入力の専用同期セット入力を使用した方がいいか
 - フリップフロップの D 入力に同期セット ロジックを使用した方がいいか
- ・ ユーザーがフリップフロップをインスタンス化すると、XST では [インスタンス化されたプリミティブの最適化 \(OPTIMIZE_PRIMITIVES\)](#) が yes の場合にのみ、同期セットが削除されます。

適用可能エレメント

次に適用します。

- ・ XST コマンド ラインからデザイン全体に適用
- ・ 特定ブロック (entity、architecture、component)
- ・ フリップフロップを表す信号
- ・ インスタンス化されたフリップフロップを表すインスタンス

適用ルール

設定したエンティティ、コンポーネント、モジュール、信号、インスタンスに適用されます。

制約値

- ・ auto (デフォルト)
- ・ yes [または true (XCF)]
- ・ no [または false (XCF)]

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute use_sync_set: string;
```

次のように指定します。

```
attribute use_sync_set of  
{entity_name|component_name|signal_name|instance_name}:  
{entity|component|signal|label} is "{auto|yes|no}";
```

Verilog の構文例

インスタンス、モジュール、または信号宣言の直前に入力します。

```
(* use_sync_set = "{auto|yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" use_sync_set={auto|yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
  
NET "signal_name" use_sync_set={auto|yes|no|true|false};  
  
END;
```

XCF の構文例 3

```
BEGIN MODEL "entity_name"  
  
INST "instance_name" use_sync_set={auto|yes|no|true|false };  
  
END;
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-use_sync_set {auto|yes|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Xilinx-Specific Options] → [Use Synchronous Set]

同期リセットの使用

同期リセットの使用 (USE_SYNC_RESET) 制約を使用すると、フリップフロップの同期リセットの使用を有効または無効にできます。

- ・ この制約は通常 ASIC のプロトタイプではオフになります。
- ・ 制約値を no にすると、最終インプリメンテーションで同期リセット リソースが使用されません。
- ・ また、デザインによっては、フリップフロップのデータ入力に同期リセット ファンクションを付けることで、ロジックが最適化され、結果が改善されることもあります。
- ・ auto に設定すると、次が比較検討されます。
 - フリップフロップ入力の専用同期リセット入力を使用した方がいいか
 - フリップフロップの D 入力に同期リセット ロジックを使用した方がいいか
- ・ ユーザーがフリップフロップをインスタンス化すると、XST では [インスタンス化されたプリミティブの最適化 \(OPTIMIZE_PRIMITIVES\)](#) が yes の場合にのみ、同期リセットが削除されます。

適用可能エレメント

次に適用します。

- ・ XST コマンド ラインからデザイン全体に適用
- ・ 特定ブロック (entity、architecture、component)
- ・ フリップフロップを表す信号
- ・ インスタンス化されたフリップフロップを表すインスタンス

適用ルール

設定したエンティティ、コンポーネント、モジュール、信号、インスタンスに適用されます。

制約値

- ・ auto (デフォルト)
- ・ yes [または true (XCF)]
- ・ no [または false (XCF)]

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute use_sync_reset: string;
```

次のように指定します。

```
attribute use_sync_reset of  
{entity_name|component_name|signal_name|instance_name}:  
is "{entity|component|signal|label; is {auto|yes|no}}";
```

Verilog の構文例

インスタンス、モジュール、または信号宣言の直前に入力します。

```
(* use_sync_reset = "{auto|yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" use_sync_reset={auto|yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
  
NET "signal_name" use_sync_reset={auto|yes|no|true|false};  
  
END;
```

XCF の構文例 3

```
BEGIN MODEL "entity_name"  
  
INST "instance_name" use_sync_reset={auto|yes|no|true|false};  
  
END;
```

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-use_sync_reset {auto|yes|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Xilinx-Specific Options] → [Use Synchronous Reset]

タイミング制約

この章では、XST のタイミング制約について説明します。

この章には、ほとんどの制約の次の情報を含めます。

- ・ 制約の説明
- ・ 適用可能エレメント
- ・ 適用ルール
- ・ 制約値
- ・ 構文例

タイミング制約の適用

- ・ タイミング制約は次のいずれかの方法で設定します。
 - [グローバル最適化目標](#)
 - [User Constraints File \(UCF\)](#)
 - [XST Constraint File \(XCF\)](#)
- ・ タイミング制約の指定方法にかかわらず、タイミング制約の処理に影響を与えるオプションは、次のとおりです。
 - [クロック信号](#)
 - [クロス クロック解析](#)
 - [タイミング制約の書き込み](#)

グローバル最適化目標を使用したタイミング制約の適用

タイミング制約は、[グローバルな最適化目標](#) (-glob_opt) コマンドライン オプションで設定できます。

- ・ グローバルな最適化目標 (-glob_opt) を使用すると、次の 5 つのグローバル タイミング制約を使用できます。
 - ALLCLOCKNETS
 - OFFSET_IN_BEFORE
 - OFFSET_OUT_AFTER
 - INPAD_TO_OUTPAD
 - MAX_DELAY
- ・ これら制約には、次のような特徴があります。
 - グローバルに適用します。
 - XST では、最適のパフォーマンスを目標として最適化が実行されるため、これらの制約に値を設定することはできません。
 - UCF ファイルで指定された制約で上書きされます。

UCF を使用したタイミング制約の適用

タイミング制約は、UCF (ユーザー制約ファイル) を使用して指定できます。

- ・ UCF ファイルからは、ネイティブ UCF 構文を使用してタイミング制約を指定できます。
- ・ XST では、次の制約がサポートされます。
 - [タイミング名 \(TNM\)](#)
 - [タイムグループ \(TIMEGROUP\)](#)
 - [周期 \(PERIOD\)](#)
 - [タイミング無視 \(TIG\)](#)
 - [FROM-TO](#)
- ・ XST では、これらの制約でワイルドカードや階層名を使用できます。

XCF を使用したタイミング制約の適用

タイミング制約は、XCF (ザイリンクス制約ファイル) を使用して指定できます。

- ・ **階層の区切り文字**にはアンダースコア (_) ではなく、スラッシュ (/) を使用してください。
- ・ XST で指定したタイミング制約のすべてまたは一部がサポートされない場合は、次が実行されます。
 - 警告メッセージが表示されます。
 - タイミング最適化段階でサポートされていないタイミング制約またはサポートされていない部分は無視されます。
- ・ [Write Timing Constraints] (`-write_timing_constraints`) プロパティを yes (チェックボックスはオン) に設定している場合は、タイミング最適化の段落で無視された制約も含め、すべての制約が最終的なネットリストに記述されます。
- ・ XCF では、次のタイミング制約がサポートされます。
 - 周期 (PERIOD)
 - オフセット (OFFSET)
 - FROM-TO
 - タイミング名 (TNM)
 - ネットのタイミング名 (TNM_NET)
 - タイムグループ (TIMEGROUP)
 - タイミング無視 (TIG)

クロック信号

クロック信号 (CLOCK_SIGNAL) 制約を使用すると、クロック信号がフリップフロップのクロック入力に接続される前に組み合わせロジックを通過する場合に、そのクロック信号を定義できます。

- ・ XST では、実際にクロック信号となる入力ピンまたは内部信号は認識できません。
- ・ クロック信号を使用して信号を定義します。

適用可能エレメント

信号に適用されます。

適用ルール

クロック信号に適用されます。

制約値

- ・ yes [または true (XCF)] (デフォルト)
- ・ no [または false (XCF)]

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

VHDL の構文例

次のように宣言します。

```
attribute clock_signal : string;
```

次のように指定します。

```
attribute clock_signal of signal_name: signal is "{yes|no}";
```

Verilog の構文例

次を信号宣言の直前に入力します。

```
(* clock_signal = "{yes|no}" *)
```

XCF の構文例

```
BEGIN MODEL "entity_name"
```

```
NET "primary_clock_signal" clock_signal={yes|no|true|false};
```

```
END;
```

クロス クロック解析

クロス クロック解析 (`-cross_clock_analysis`) コマンドライン オプションを使用すると、XST で複数のクロックドメイン間のタイミング最適化を実行できます。

- ・ タイミング最適化は常に必要なわけではないので、デフォルトではオフになっています。最適化がオフになっている場合、XST は各クロックドメイン内でのみタイミングを最適化します。
- ・ フリップフロップのリタイミングを使用するために [レジスタの自動調整 \(REGISTER_BALANCING\)](#) を使用した場合は、クロス クロック解析制約でもリタイミングの範囲が定義されます。
 - クロス クロック解析がオンになっている場合、必要であれば 1 つのクロックドメインから別のドメインにロジックが移動されることもあります。
 - クロス クロック解析がオフになっている場合は、レジスタの自動調整が各クロックドメイン内でのみ実行されます。
- ・ クロックドメイン間のタイミング情報は、デフォルトで[合成レポート](#)に含まれます。クロックドメイン間の最適化を、このレポートを入手するためにアクティベートする必要はありません。詳細は、「[クロス クロックドメイン タイミング情報の取得](#)」を参照してください。

適用可能エレメント

XST コマンドラインでデザイン全体に適用されます。

適用ルール

ありません。

制約値

- ・ yes
- ・ no

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「[はじめに](#)」の構文例を参照してください。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
-cross_clock_analysis {yes|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックスの [Synthesis Options] ページにある [Cross Clock Analysis]

FROM-TO

2 つのグループ間のタイミング制約を設定します。

この場合のグループは、ユーザー定義のグループまたは定義済みのグループです。

- ・ FF
- ・ PAD
- ・ RAM

詳細は、『[制約ガイド](#)』(UG625) を参照してください。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。
詳細は、『[はじめに](#)』の構文例を参照してください。

XCF の構文例

```
TIMESPEC TSname = FROM group1 TO group2 value;
```

グローバル最適化目標

グローバルな最適化目標 (-glob_opt) コマンドライン オプションを使用すると、次が実現可能です。

- ・ 最適なパフォーマンスにするため、でのデザイン全体の最適化方法を定義します。
- ・ XST で次のデザイン領域を最適化できるようにします。
 - レジスタからレジスタ
 - 入力パッドからレジスタ
 - レジスタから出力パッド
 - 入力パッドから出力パッド

グローバルなタイミング制約

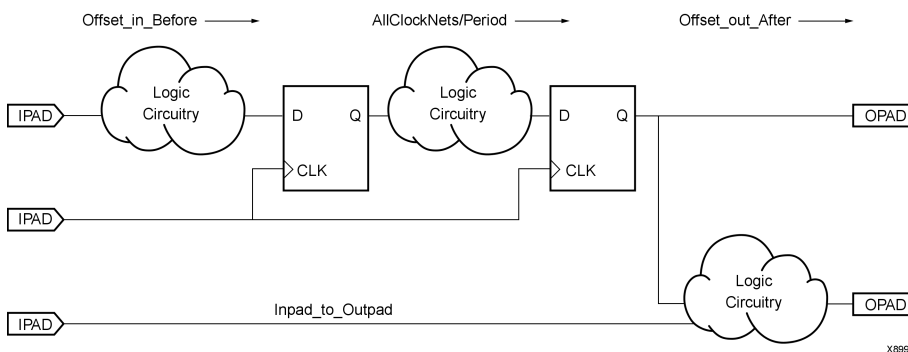
- ・ 次のグローバル タイミング制約のいずれかを選択できます。
 - ALLCLOCKNETS
デザイン全体の周期を最適化します。
 - OFFSET_BEFORE
特定クロックまたはデザイン全体に対して、入力パッドからクロックまでの最大遅延を最適化します。
 - OFFSET_OUT_AFTER
特定クロックまたはデザイン全体に対して、クロックから出力パッドまでの最大遅延を最適化します。
 - INPAD_OUTPAD
デザイン全体の入力パッドから出力パッドまでの最大遅延を最適化します。
 - MAX_DELAY
上記 4 つの制約を統合したものです。
- ・ これらのグローバル タイミング制約には、次のような特徴があります。
 - デザイン全体にグローバルに適用されます。
 - 制約ファイルでタイミング制約が指定されていない場合にのみ適用します。
 - XST では、最適のパフォーマンスを目標として最適化が実行されるため、ユーザー指定の値を設定することはできません。
 - UCF ファイルで指定された制約で上書きされます。

グローバル最適化のドメインの定義

指定できるドメインは次のとおりです。

- ・ ALLCLOCKNETS (レジスタからレジスタ)
デザイン内のすべてのクロックに対し、同じクロックパス上にあるレジスタ間のすべてのパスが指定されます。クロックドメイン遅延を考慮に入れるには、[クロスクロック解析 \(-cross_clock_analysis\)](#) を yes に設定します。
- ・ OFFSET_IN_BEFORE (入力パッドからレジスタ)
プライマリ入力ポートからすべての順次エレメント、または指定したクロック信号名で駆動される特定の順次エレメントまでのパスが指定されます。
- ・ OFFSET_OUT_AFTER (レジスタから出力パッド)
OFFSET_IN_BEFORE と同様ですが、順次エレメントからの制約をすべてのプライマリ出力ポートに設定します。
- ・ INPAD_TO_OUTPAD (入力パッドから出力パッド)
最大組み合わせパス制約が指定されます。
- ・ MAX_DELAY
 - ALLCLOCKNETS
 - OFFSET_IN_BEFORE
 - OFFSET_OUT_AFTER
 - INPAD_TO_OUTPAD

グローバル最適化目標のドメインの図



構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンド ライン構文例

run コマンドでグローバルに設定します。

```
glob_opt
{allclocknets|offset_in_before|offset_out_after|inpad_to_outpad|max_delay}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process] → [Process Properties] → [Synthesis Options] → [Global Optimization Goal]

オフセット

オフセット (OFFSET) 制約では、外部クロックと関連するデータ入力ピンまたはデータ出力ピンとのタイミング関係を指定します。

OFFSET (オフセット) 制約には、次の特徴があります。

- タイミング制約です。
- パッド関連の信号にのみ使用されます。
- デザインの内部信号への信号到着時間は延長できません。
- 外部ネットからデータ入力とクロック入力が供給されるフリップフロップで、セットアップ タイムの要件が満たされているかを計算できます。
- 外部デバイス ピンをクロック ソースとする内部フリップフロップの Q 出力から生成された外部出力ネットの遅延を指定

この制約の詳細は、『[制約ガイド](#)』(UG625)を参照してください。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、『[はじめに](#)』の構文例を参照してください。

XCF の構文例

```
OFFSET = {IN|OUT} offset_time [units] {BEFORE|AFTER} clk_name  
[TIMEGRP group_name];
```


周期

PERIOD は、基本的なタイミング制約および合成制約です。

- ・ PERIOD 制約を指定すると、デスティネーション エLEMENT のグループで定義されているクロックドメイン内で、すべての同期ELEMENT間のタイミングが確認されます。クロックが別のクロックの関数として定義されている場合、グループには複数のクロックドメインを通過するパスが含まれます。
- ・ MMCM ブロックの場合は、PERIOD 制約を該当するクロック入力信号に適用します。
 - MMCM クロック出力信号に対して関連する PERIOD 制約を手動で作成する必要はありません。
 - PERIOD 制約は XST で自動的に設定されるので、正確なタイミング データに基づいて合成が指定されます。
- ・ これらの PERIOD 制約は NGC ネットリストで指定した場合にのみ書き出されます。
 - 派生した制約はこの時点では書き出されず、
 - 後のインプリメンテーション中に書き出されます。

この制約の詳細は、『[制約ガイド](#)』(UG625)を参照してください。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、『[はじめに](#)』の構文例を参照してください。

XCF の構文例

```
NET netname PERIOD = value [{HIGH|LOW} value];
```

タイミング名

タイミング名 (TNM) 制約では、タイミング仕様で使用されるグループを構成するインスタンスを指定できます。

- ・ TNM は基本的なグループ制約です。
- ・ 次の特定のエレメントをグループ化することにより、タイミング仕様の適用を簡略化できます。
 - FF
 - RAM
 - LATCH
 - PAD
 - BRAM_PORTA
 - BRAM_PORTB
 - CPU
 - HSIO
 - MULT
- ・ TNM では、RISING および FALLING キーワードがサポートされます。

この制約の詳細は、『[制約ガイド](#)』(UG625)を参照してください。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「[はじめに](#)」の構文例を参照してください。

XCF の構文例

```
{INST|NET|PIN} inst_net_or_pin_name TNM =  
[predefined_group:] identifier;
```

ネットのタイミング名

ネットのタイミング名 (TNM_NET) 制約は、入力パッドの場合を除き、ネットに設定した**タイミング名 (TNM)** と基本的に同じです。

- ・ TNM_NET は、通常特定ネットを指定するために使用されます。TNM_NET で指定されたダウンストリームの同期エレメントおよびパッドは、すべて 1 つのグループと見なされます。
- ・ **タイミング名 (TNM)** および TNM_NET に**PERIOD 制約**を付けて次に使用する場合は、特別なルールが適用されます。
 - DLL
 - DCM
 - PLL

詳細は、『制約ガイド』の「CLKDLL、DCM、PLL での PERIOD 指定」を参照してください。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XCF の構文例

```
NET netname TNM_NET = [predefined_group:] identifier;
```

タイムグループ

TIMEGRP は、基本的なグループ制約です。

- ・ TIMEGRP は次のために使用します。
 - タイミング名 (TNM) 識別子を使用してグループを命名
 - ほかのグループに相対的にグループを指定
 - 既存グループを組み合わせたグループを作成
 - タイムグループ制約は、次で適用できます。
 - ◆ XCF (ザイリンクス制約ファイル)、または
 - ◆ NCF (ネットリスト制約ファイル)
- ・ TIMEGRP 属性を使用して次のいずれかの方法でグループを作成します。
 - 複数のグループを 1 つのグループにまとめる
 - クロックの立ち上がり/立ち下がりによってフリップフロップのサブグループを指定する

この制約の詳細は、『[制約ガイド](#)』(UG625)を参照してください。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、『[はじめに](#)』の構文例を参照してください。

XCF の構文例

```
TIMEGRP newgroup = existing_grp1 existing_grp2 [existing_grp3  
...];
```

タイミング無視

タイミング無視 (TIG) 制約には、次の特徴があります。

- ・ タイミング解析および最適化の際に、制約を設定したネットを通過するパスが無視されます。
- ・ 無視する信号の名前に適用します。

この制約の詳細は、『[制約ガイド](#)』(UG625)を参照してください。

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、『[はじめに](#)』の構文例を参照してください。

XCF の構文例

```
NET net_name TIG;
```

タイミング制約の書き込み

タイミング制約の書き込み (`-write_timing_constraints`) コマンドライン オプションでは、NGC ファイルへタイミング制約が書き込まれるタイミングを指定できます。

- ・ タイミング制約は自動的に NGC ファイルに書き込まれません。
- ・ タイミング制約は、次の設定をしている場合にのみ NGC ファイルに書き込まれます。
 - ISE® Design Suite で [Write Timing Constraints] をオン
[Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックス → [Synthesis Options] → [Write Timing Constraints]
 - `-write_timing_constraints` コマンドライン オプションを使用

適用可能エレメント

XST コマンドラインでデザイン全体に適用されます。

適用ルール

ありません。

制約値

- ・ **yes**
- ・ **no** (デフォルト)

構文例

ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。詳細は、「はじめに」の構文例を参照してください。

XST コマンドライン構文例

run コマンドでグローバルに設定します。

```
-write_timing_constraints {yes|no}
```

ISE Design Suite での設定

ISE Design Suite でグローバルに定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Write Timing Constraints]

サードパーティの制約

XST では、多くのサードパーティ制約がサポートされます。

- ・ 次の表は、それらの制約と同じ動作をする XST 制約を示しています。
 - 次の表で「あり」になっている制約は、完全にサポートされています。
 - 部分的にのみサポートされる場合は、その詳細が次の表の自動識別の列に記述されています。
 - 各制約の機能などについては、該当するベンダーのマニュアルを参照してください。
- ・ サードパーティ制約の適用方法は、ザイリンクス制約と同じです。
 - VHDL 属性
 - Verilog 属性
 - XCF 制約

VHDL でのサードパーティ制約

- ・ VHDL では、標準の属性構文を使用します。
- ・ HDL ソース コードに変更を加える必要はありません。

Verilog でのサードパーティ制約

- ・ サードパーティのメタ コメント構文を含む Verilog の場合、そのメタ コメント構文は XST の命名規則に従うように変更する必要があります。
- ・ 制約名とその値は、サードパーティ ツールで表示されているとおりに使用できます。
- ・ Verilog -2001 属性の場合、HDL コードを変更する必要はありません。制約は自動的に VHDL 属性構文に変換されます。

サードパーティの制約と同等の XST 制約

名前	ベンダー	同等の XST 制約	自動認識	HDL
black_box	Synopsys	ボックス タイプ	なし	VHDL Verilog
black_box_pad_pin	Synopsys	なし	なし	なし
black_box_tri_pins	Synopsys	なし	なし	なし
cell_list	Synopsys	なし	なし	なし
clock_list	Synopsys	なし	なし	なし
enum	Synopsys	なし	なし	なし
full_case	Synopsys	フル ケース	なし	Verilog
ispad	Synopsys	なし	なし	なし
map_to_module	Synopsys	なし	なし	なし
net_name	Synopsys	なし	なし	なし
parallel_case	Synopsys	パラレル ケース	なし	Verilog
return_port_name	Synopsys	なし	なし	なし
resource_sharing directives	Synopsys	リソース共有	なし	VHDL Verilog
set_dont_touch_network	Synopsys	必要なし	なし	なし
set_dont_touch	Synopsys	必要なし	なし	なし
set_dont_use_cel_name	Synopsys	必要なし	なし	なし
set_prefer	Synopsys	なし	なし	なし
state_vector	Synopsys	なし	なし	なし
syn_allow_retiming	Synopsys	レジスタ自動調整	なし	VHDL Verilog
syn_black_box	Synopsys	ボックス タイプ	あり	VHDL Verilog
syn_direct_enable	Synopsys	なし	なし	なし
syn_edif_bit_format	Synopsys	なし	なし	なし
syn_edif_scalar_format	Synopsys	なし	なし	なし
syn_encoding	Synopsys	FSM Encoding Algorithm	あり (safe は自動認識されません。 XST でセーフ インプリメンテーションを使用し てこのモードを有効にしてください)	VHDL Verilog
syn_enum_encoding	Synopsys	列挙型エンコード手法	なし	VHDL

名前	ベンダー	同等の XST 制約	自動認識	HDL
syn_hier	Synopsys	階層の維持	あり syn_hier = hard は keep_hierarchy = soft として認識 syn_hier = remove は keep_hierarchy = no として認識 XST でサポートされる syn_hier の値は自動認識で hard と remove のみ	VHDL
				Verilog
syn_isclock	Synopsys	なし	なし	なし
syn_keep	Synopsys	キープ	あり	VHDL
				Verilog
syn_maxfan	Synopsys	最大ファンアウト	あり	VHDL
				Verilog
syn_netlist_hierarchy	Synopsys	ネットリスト階層	なし	VHDL
				Verilog
syn_noarrayports	Synopsys	なし	なし	なし
syn_noclockbuf	Synopsys	バッファ タイプ	あり	VHDL
				Verilog
syn_noprune	Synopsys	インスタンス化されたプリミティブの最適化	あり	VHDL
				Verilog
syn_pipeline	Synopsys	レジスタ自動調整	なし	VHDL
				Verilog
syn_preserve	Synopsys	等価レジスタの削除	Yes	VHDL
				Verilog
syn_ramstyle	Synopsys	RAM 抽出 および RAM スタイル	あり 指定してもしなくても no_rw_check モードでインプリメントします。 area 値は無視されます。	VHDL
				Verilog
syn_reference_clock	Synopsys	なし	なし	なし
syn_replicate	Synopsys	レジスタの複製	あり	VHDL
				Verilog
syn_romstyle	Synopsys	ROM 抽出 および ROM スタイル	Yes	VHDL
				Verilog
syn_sharing	Synopsys	リソース共有	なし	VHDL
				Verilog

名前	ベンダー	同等の XST 制約	自動認識	HDL
syn_state_machine	Synopsys	FSM 自動抽出	あり	VHDL
				Verilog
syn_tco	Synopsys	なし	なし	なし
syn_tpd	Synopsys	なし	なし	なし
syn_tristate	Synopsys	なし	なし	なし
syn_tristatetomux	Synopsys	なし	なし	なし
syn_tsu	Synopsys	なし	なし	なし
syn_useenables	Synopsys	クロック イネーブルの使用	なし	なし
syn_useioff	Synopsys	I/O レジスタの IOB 内へのバック (IOB)	なし	VHDL
				Verilog
synthesis_translate_off	Synopsys	Translate Off と Translate On	Yes	VHDL
synthesis_translate_on	Synopsys			Verilog
xc_alias	Synopsys	なし	なし	なし
xc_clockbuftype	Synopsys	バッファ タイプ	なし	VHDL
				Verilog
xc_fast	Synopsys	FAST	なし	VHDL
				Verilog
xc_fast_auto	Synopsys	FAST	なし	VHDL
				Verilog
xc_global_buffers	Synopsys	BUFG (XST)	なし	VHDL
				Verilog
xc_ioff	Synopsys	I/O レジスタの IOB 内へのバック	なし	VHDL
				Verilog
xc_isgsr	Synopsys	なし	なし	なし
xc_loc	Synopsys	LOC	あり	VHDL
				Verilog
xc_map	Synopsys	Map Entity on a Single LUT	あり	VHDL
			XST でサポートされる値は lut のみです	Verilog
xc_ncf_auto_relax	Synopsys	なし	なし	なし
xc_nodelay	Synopsys	NODELAY	なし	VHDL
				Verilog
xc_padtype	Synopsys	I/O 規格	なし	VHDL
				Verilog
xc_props	Synopsys	なし	なし	なし

名 前	ベンダー	同等の XST 制約	自動認識	HDL
xc_pullup	Synopsys	PULLUP	なし	VHDL
				Verilog
xc_rloc	Synopsys	RLOC	Yes	VHDL
				Verilog
xc_fast	Synopsys	FAST	なし	VHDL
				Verilog
xc_slow	Synopsys	なし	なし	なし
xc_uset	Synopsys	U_SET	Yes	VHDL
				Verilog

合成レポート

- ・ 合成レポートには、次のような特徴があります。
 - ASCII テキスト ファイル
 - レポートとログの混在したファイルです。
 - XST 合成の実行に関する情報を含みます。
- ・ 合成中は、合成レポートを使用すると次が可能になります。
 - 合成の進捗状況を制御
 - 暫定的な合成結果を確認
- ・ 合成後に合成レポートを使用すると、次を判断できるようになります。
 - HDL 記述が予測どおりに処理されたかどうか
 - 合成されたネットリストがインプリメンテーションまで実行されると、デバイス使用率と最適化レベルがデザイン目標を満たしそうかどうか

合成レポートの内容

合成レポートには、次のセクションが含まれます。

- ・ [目次](#)
- ・ [合成オプションのサマリ](#)
- ・ [HDL 解析およびエラボレーション](#)
- ・ [HDL 合成](#)
- ・ [アドバンス HDL 合成](#)
- ・ [下位合成](#)
- ・ [パーティション レポート](#)
- ・ [Design Summary](#)

目次

目次を使用すると、レポートの該当セクションをすぐに表示できます。詳細は、「[合成レポートのナビゲーション](#)」を参照してください。

合成オプションのサマリ

Synthesis Options Summary セクションには、現在の合成に使用されたパラメーターやオプションがまとめられています。

HDL 解析およびエラボレーション

HDL 解析およびエラボレーション中、XST では次が実行されます。

- ・ 合成プロジェクトを構成する VHDL および Verilog ファイルの解析
- ・ ファイルの内容の解釈
- ・ デザイン階層の認識
- ・ HDL コードのミスの表示
- ・ 次のような潜在的な問題を指摘
 - 合成後と HDL のシミュレーションの不一致
 - 潜在的なマルチソースの状態

合成の後半で問題が発生すると、このセクションにその問題の原因が表示されます。

HDL の合成

HDL の合成中、XST では次が実行されます。

- ・ 後でデバイス用のインプリメンテーションが可能な次の基本的なマクロが認識されます。
 - レジスタ
 - 加算器
 - 乗算器
- ・ FSM の記述をブロックごとに検索します。
- ・ 推論したマクロの統計を含めた HDL 合成レポートを生成します。

マクロ処理および合成プロセス中に表示されるメッセージの詳細については第 7 章「HDL のコーディング手法」を参照してください。

アドバンス HDL 合成レポート

アドバンス HDL 合成中、XST では HDL 合成中に推論された基本的なマクロがより大きなマクロにまとめられます。

- ・ マクロ ブロックには、次が含まれます。
 - カウンター
 - パイプライン乗算器
 - 乗累算ファンクション
- ・ 推論された各有限ステート マシン (FSM) に選択したエンコード方法に関するレポートが表示されます。
- ・ アドバンス HDL 合成レポートでは、デザイン全体で認識されたマクロのサマリが記述されます。
- ・ この認識されたマクロは、タイプ別に表示されます。

詳細は、第 7 章「HDL コーディング手法」を参照してください。

下位レベルの合成

Low Level Synthesis セクションには、次を含む XST で実行された下位レベルの最適化に関する情報が表示されます。

- ・ 等価フリップフロップの削除
- ・ 定数フリップフロップの最適化
- ・ レジスタの複製

パーティション レポート

パーティション レポートには、デザイン パーティションに関する情報が表示されます。

デザイン サマリ (Design Summary)

Design Summary セクションからは、次の点を確認できます。

- ・ 合成が問題なく実行されたかどうか
- ・ デバイス使用率と回路パフォーマンスがデザイン目標を満たしたかどうか

このセクションでは、次の内容が説明されています。

- ・ プリミティブおよびブラック ボックスの使用率
- ・ デバイス使用率のサマリ
- ・ パーティション リソース サマリ
- ・ タイミング レポート
- ・ クロック情報
- ・ 非同期制御信号の情報
- ・ タイミング サマリ
- ・ タイミングの詳細
- ・ 暗号化されたモジュール

プリミティブおよびブラック ボックスの使用率 (Primitive and Black Box Usage)

Primitive and Black Box Usage サブセクションには、次の使用統計が表示されます。

- ・ デバイス プリミティブ
- ・ 識別されたブラック ボックス

プリミティブは、次のグループに分類されます。

- ・ BEL
- ・ LUT、MUXCY、XORCY、MUXF5、MUXF6 などの基本的な論理プリミティブすべて
- ・ フリップフロップおよびラッチ
- ・ ブロックおよび分散 RAM
- ・ シフトレジスタ プリミティブ
- ・ トライステート バッファ
- ・ クロック バッファ
- ・ I/O バッファ
- ・ AND2 や OR2 などのさらに複雑なその他の論理プリミティブ
- ・ その他のプリミティブ

デバイス使用率のサマリ (Device Utilization Summary)

Device Utilization Summary には、次のようなファンクションのデバイス使用率の概算が表示されます。

- ・ スライス ロジックの使用率
- ・ スライス ロジックの分配
- ・ フリップフロップ数
- ・ I/O 使用率
- ・ ブロック RAM 数
- ・ DSP ブロック数

MAP を後で実行したときに生成されるレポートと類似しています。

パーティション リソース サマリ (Partition Resource Summary)

パーティションが定義されると、このサブセクションに Device Utilization Summary (デバイス使用率のサマリ) と同じような情報がパーティションごとの基準で表示されます。

タイミング レポート

Timing Report サブセクションには、タイミング概算が表示され、この情報を次に利用することができます。

- ・ デザインがパフォーマンスおよびタイミング要件を満たしているかどうかの判断
- ・ パフォーマンスおよびタイミング要件が達成できない場合のボトルネックの検出

クロック情報 (Clock Information)

Clock Information サブセクションには、次の情報が表示されます。

- ・ クロック数
- ・ 各クロックのバッファ方法
- ・ 対応するファンアウト

Clock Information サブセクションの例

Clock Information:

Clock Signal	Clock buffer (FF name)	Load
CLK	BUFGP	11

非同期制御信号の情報 (Asynchronous Control Signals Information)

Asynchronous Control Signals Information サブセクションには、次の情報が表示されます。

- ・ 非同期セット/リセット数
- ・ 各信号のバッファを介する方法
- ・ 対応するファンアウト

Asynchronous Control Signals Report Information サブセクションの例

Asynchronous Control Signals Information:

Control Signal	Buffer (FF name)	Load
rstint (MACHINE/current_state_Out01:0)	NONE (sixty/lsbcount/qoutsig_3)	4
RESET	IBUF	3
sixty/msbclr (sixty/msbclr:0)	NONE (sixty/msbcount/qoutsig_3)	4

タイミング サマリ

Timing Summary サブセクションには、次のネットリストの 4 つの可能性のあるクロックドメインすべてに関するタイミング情報が表示されます。

- ・ 最小周期
レジスタ間のパス
- ・ クロック前の最小入力到着時間
入力からレジスタへのパス
- ・ クロック後の最大出力必須時間
レジスタからへのパス
- ・ 最大組み合わせパス遅延
入力パッドから出力パッドへのパス

このタイミング情報は、概算です。正確なタイミング情報については、配置配線後に生成される TRACE レポートを参照してください。

Timing Summary サブセクションの例

Timing Summary:

Speed Grade: -1

Minimum period: 2.644ns (Maximum Frequency: 378.165MHz)

Minimum input arrival time before clock: 2.148ns

Maximum output required time after clock: 4.803ns

Maximum combinatorial path delay: 4.473ns

タイミングの詳細

Timing Details サブセクションには、次を含む各クロック領域で最もクリティカルなパスに関する情報が表示されます。

- ・ 開始点
- ・ 終了点
- ・ 最大遅延
- ・ ロジック レベル
- ・ パスを個々のネットおよびコンポーネント遅延にまで細かく分解した情報
- ・ ネット ファンアウトに関する情報
- ・ 配線とロジック間の分配

パス分割

XST で [ネットリスト階層 \(-netlist_hierarchy\)](#) が使用されて階層ネットリストが生成される際、レポートされるパスが階層バウンダリを超えていることがあります。

この場合、分割されたパスでは、begin scope および end scope キーワードでそのパスの階層への出入りが示されます。

Timing Details レポートの例

Timing Details:

All values displayed in nanoseconds (ns)

=====

Timing constraint: Default period analysis for Clock 'CLK'

Clock period: 2.644ns (frequency: 378.165MHz)

Total number of paths / destination ports: 77 / 11

Delay: 2.644ns (Levels of Logic = 3)

Source: MACHINE/current_state_FFd3 (FF)

Destination: sixty/msbcount/qoutsig_3 (FF)

Source Clock: CLK rising

Destination Clock: CLK rising

Data Path: MACHINE/current_state_FFd3 to sixty/msbcount/qoutsig_3

		Gate	Net	
Cell:in->out	fanout	Delay	Delay	Logical Name (Net Name)

FDC:C->Q	8	0.272	0.642	ctrl/state_FFd3 (ctrl/state_FFd3)
LUT3:I0->O	3	0.147	0.541	Ker81 (clkenable)
LUT4_D:I1->O	1	0.147	0.451	sixty/msbce (sixty/msbce)
LUT3:I2->O	1	0.147	0.000	sixty/msbcount/qoutsig_3_rstpot (N43)
FDC:D		0.297		sixty/msbcount/qoutsig_3

Total		2.644ns (1.010ns logic, 1.634ns route)		
		(38.2% logic, 61.8% route)		

タイミング制約のデフォルトのパス解析レポート例

Timing constraint: Default path analysis

Total number of paths / destination ports: 36512 / 16

Delay: 4.326ns (Levels of Logic = 14)

Source: a<0> (PAD)

Destination: out<3> (PAD)

Data Path: a<0> to out<>

Cell:in->out	fanout	Gate		Net	
		Delay	Delay	Logical Name	(Net Name)
IBUF:I->O	5	0.003	0.376	a_0_IBUF	(a_0_IBUF)
begin scope: 'm'					
begin scope: 'a1'					
LUT2:I0->O	1	0.053	0.000	Madd_out_Madd_lut<0>	(Madd_out_Madd_lut<0>)
MUXCY:S->O	1	0.219	0.000	Madd_out_Madd_cy<0>	(Madd_out_Madd_cy<0>)
MUXCY:CI->O	1	0.015	0.000	Madd_out_Madd_cy<1>	(Madd_out_Madd_cy<1>)
MUXCY:CI->O	1	0.015	0.000	Madd_out_Madd_cy<2>	(Madd_out_Madd_cy<2>)
MUXCY:CI->O	1	0.015	0.000	Madd_out_Madd_cy<3>	(Madd_out_Madd_cy<3>)
MUXCY:CI->O	1	0.015	0.000	Madd_out_Madd_cy<4>	(Madd_out_Madd_cy<4>)
MUXCY:CI->O	1	0.015	0.000	Madd_out_Madd_cy<5>	(Madd_out_Madd_cy<5>)
MUXCY:CI->O	0	0.015	0.000	Madd_out_Madd_cy<6>	(Madd_out_Madd_cy<6>)
XORCY:CI->O	1	0.180	0.279	Madd_out_Madd_xor<7>	(out<7>)
end scope: 'a1'					
DSP48E1:A7->P2	1	2.843	0.279	Maddsub_out	(out_2_OBUF)
end scope: 'm'					
OBUF:I->O		0.003		out_2_OBUF	(out<2>)
Total					
		4.326ns (3.391ns logic, 0.935ns route)			
		(78.4% logic, 21.6% route)			

クロス クロックドメイン タイミング情報の取得

- ・ 「Cross Domains Crossing Report」セクションには、次の特徴があります。
 - クロックドメイン クロス (CDC) パスをレポートします。
 - デフォルトで含まれます。
 - 「Timing Details」セクションの後に記述されます。
 - XST によるクロス クロックドメインの最適化の実行に関係なく表示されます。
 - XCF (ザイリンクス制約ファイル) でのタイミング制約の指定に関係なく表示されます。
- ・ クロス クロックドメインのタイミング情報を取得するのに、[クロス クロック解析 \(-cross_clock_analysis\)](#) をオンにする必要はありません。
- ・ [クロス クロック解析 \(-cross_clock_analysis\)](#) は、クロックドメイン間のタイミング最適化を達成する場合にのみ使用します。

クロスドメイン クロッシング レポートの例

Clock Domains Crossing Report:

Clock to Setup on destination clock clk2

	Src:Rise	Src:Fall	Src:Rise	Src:Fall
Source Clock	Dest:Rise	Dest:Rise	Dest:Fall	Dest:Fall
clk1	0.804			
clk2	0.661			

Clock to Setup on destination clock clk3

	Src:Rise	Src:Fall	Src:Rise	Src:Fall
Source Clock	Dest:Rise	Dest:Rise	Dest:Fall	Dest:Fall
clk2			0.809	
clk3			0.651	

暗号化されたモジュール (Encrypted Modules)

XST では、暗号化されたモジュールに関する情報はすべて表示しません。

合成レポートのナビゲート

合成レポートでナビゲートするには、次の方法を使用します。

- ・ ISE® Design Suite のレポート ナビゲーション
- ・ コマンド ライン モードのレポート ナビゲーション

ISE Design Suite レポートのナビゲーション

ISE® Design Suite を使用する場合、XST では SYR (.syr) ファイルが生成されます。

SYR ファイルとは、次のようなファイルです。

- ・ 合成レポートすべてを含有
- ・ ISE Design Suite のプロジェクトのあるディレクトリにあり
- ・ 合成レポートのさまざまなセクションをナビゲーション ペインからナビゲート可能

コマンド ライン モードのレポート ナビゲーション

XST では、コマンド ライン モードで SRP (.srp) ファイルが生成されます。

- ・ SRP ファイルは、合成レポートすべてを含んだ ASCII 形式のテキスト ファイルです。
- ・ SRP ファイルの目次には、リンクが付いていませんので、[Find] を使用してナビゲートします。

合成レポートの情報

次を使用すると、合成レポートに表示される情報を削減できます。

- ・ メッセージ フィルター
- ・ Quiet モード
- ・ Silent モード

メッセージ フィルター

ISE® Design Suite でメッセージ フィルターを使用すると、合成レポートに特定のメッセージのみを表示できます。

- ・ 個々のメッセージまたはカテゴリ別にメッセージをフィルターできます。
- ・ 詳細は、ISE Design Suite ヘルプの「メッセージ フィルターの使用」を参照してください。

Quiet モード

- ・ 通常は、画面 (stdout) にすべてのログが出力されます。Quiet モードを使用すると、コンピュータの画面 (stdout) に表示されるメッセージの量を制限できます。
- ・ Quiet モードによって、合成レポートが変更されることはありません。XST 合成レポートには、合成情報がすべてフィルターのかかっていない状態で表示されます。
- ・ このモードを設定するには、-intstyle オプションを次のいずれかにします。

オプション	メッセージのフォーマット
ise	ISE® Design Suite
xflow	XFLOW

コンピューター画面に表示されるレポート セクション

Quiet モードを使用してコンピューター画面に表示される XST 合成レポートのセクションは、次のとおりです。

- ・ [デバイス使用率のサマリ](#)
- ・ [クロック情報](#)
- ・ [タイミング サマリ](#)

コンピューター画面に表示されないレポート セクション

Quiet モードを使用してコンピューター画面に表示されない XST 合成レポートのセクションは、次のとおりです。

- ・ 著作権情報
- ・ [目次](#)
- ・ [合成オプションのサマリ](#)
- ・ [デザイン サマリ \(Design Summary\)](#) セクションには、次が記述されます。
 - 最終結果のセクション
 - タイミング数値が合成における概算にすぎないことを示す注記
 - タイミングの詳細
 - CPU (XST ランタイム)
 - メモリ 使用率

Silent モード

Silent モードを使用すると、コンピューターの画面 (stdout) にメッセージが表示されないようになります。

- ・ [合成レポート](#)は、ログ ファイルに記述されます。
- ・ このモードを設定するには、`-intstyle` オプションを `silent` にします。

命名規則

合成ツールは、命名方法に従ってオブジェクトを命名し、合成済みネットリストに書き込みます。

- ・ 命名規則は、次のようになっている必要があります。
 - 論理的
 - 一貫性あり
 - 予測可能
 - 繰り返し可能
- ・ 命名規則に従うと、次が実現できます。
 - 制約を使用してデザインのインプリメンテーションを制御
 - タイミング クロージャ サイクルを削減

命名規則のコード例

アップデート情報は、「はじめに」のコード例を参照してください。

always ブロック (ラベルあり) の reg を示す Verilog コード例

```
//  
// A reg in a labelled always block  
//  
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip  
// File: Naming_Conventions/reg_in_labelled_always.v  
//  
module top (  
    input  clk,  
    input  di,  
    output do  
);  
  
    reg data;  
  
    always @(posedge clk)  
    begin : mylabel  
  
        reg tmp;  
  
        tmp <= di;           // Post-synthesis name : mylabel.tmp  
        data <= ~tmp;        // Post-synthesis name : data  
  
    end  
  
    assign do = ~data;  
  
endmodule
```

if-generate 文 (ラベルなし) でプリミティブ インスタンスエーションを記述した Verilog コード例

```
//  
// A primitive instantiation in a if-generate without label  
//  
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip  
// File: Naming_Conventions/if_generate_nolabel.v  
//  
module top (  
    input  clk,  
    input  di,  
    output do  
);  
  
    parameter TEST_COND = 1;  
  
    generate  
  
        if (TEST_COND) begin  
            FD myinst (.C(clk), .D(di), .Q(do)); // Post-synthesis name : myinst  
        end  
  
    endgenerate  
  
endmodule
```

if-generate 文 (ラベルあり) でプリミティブ インスタンスエーションを記述した Verilog コード例

```
//  
// A primitive instantiation in a labelled if-generate  
//  
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip  
// File: Naming_Conventions/if_generate_label.v  
//  
module top (  
    input  clk,  
    input  rst,  
    input  di,  
    output do  
);  
  
    // parameter TEST_COND = 1;  
    parameter TEST_COND = 0;  
  
    generate  
  
        if (TEST_COND)  
            begin : myifname  
                FDR myinst (.C(clk), .D(di), .Q(do), .R(rst));  
                // Post-synthesis name : myifname.myinst  
            end  
        else  
            begin : myelsenname  
                FDS myinst (.C(clk), .D(di), .Q(do), .S(rst));  
                // Post-synthesis name : myelsenname.myinst  
            end  
  
    endgenerate  
  
endmodule
```

process 文 (ラベルあり) の変数を示す VHDL コード例

```
--
-- A variable in a labelled process
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: Naming_Conventions/var_in_labelled_process.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity top is
  port (
    clk : in std_logic;
    di  : in std_logic;
    do  : out std_logic
  );
end top;

architecture behavioral of top is
  signal data : std_logic;
begin

  mylabel: process (clk)
    variable tmp : std_logic;
  begin
    if rising_edge(clk) then
      tmp := di;          -- Post-synthesis name : mylabel.tmp
    end if;
    data <= not(tmp);
  end process;

  do <= not(data);

end behavioral;
```

ブール型で記述したフリップフロップの VHDL コード例

```
--
-- Naming of boolean type objects
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: Naming_Conventions/boolean.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity top is
    port(
        clk : in  std_logic;
        di   : in  boolean;
        do   : out boolean
    );
end top;

architecture behavioral of top is
    signal data : boolean;
begin

    process (clk)
    begin
        if rising_edge(clk) then
            data <= di;      -- Post-synthesis name : data
        end if;
    end process;

    do <= not(data);

end behavioral;
```

ネットの命名規則

XST では、ネット名が次の規則に基づいて作成されます（命名規則は、優先順にリストしています）。

1. 外部ピン名を保持します。
2. 信号名で階層を保持します。
階層区切り文字は、[Hierarchy Separator] (`-hierarchy_separator`) で定義します。XST では、デフォルトの階層区切り文字はスラッシュ (/) です。
3. ステートビットを含むレジスタの出力信号名を保持します。
レジスタが推論されるレベルからの階層名を使用します。
4. クロック バッファの出力信号名には、クロック信号名の後にアンダースコアとクロック バッファ タイプ (`_BUFGP`、`_IBUFG` など) を付けます。
5. レジスタおよびトライステート名に対する入力ネットを保持します。
6. プリミティブおよびブラック ボックスに接続されている信号名を保持します。
7. IBUF の出力ネット名は、`<signal_name>IBUF` のようになります。
たとえば、IBUF 出力が DIN 信号を駆動する場合、この IBUF の出力ネットは `DIN_IBUF` になります。
8. OBUF に対する入力ネットの名前は、`<signal_name>OBUF` のようになります。
たとえば、OBUF 入力 が DOUT 信号で駆動される場合、この OBUF の入力ネットは `DOUT_OBUF` になります。
9. 内部 (組み合わせ) ネットの名前のベース名には、ユーザーの HDL 信号名が使用されます。
10. バスを拡張した場合のネットは、次のような形式になります。
`<bus_name><left_delimiter><position>#<right_delimiter>`
 - ・ デフォルトの左区切り文字は `<`、右区切り文字は `>` です。
 - ・ これを変更するには、[バスの区切り文字指定 \(-bus_delimiter\)](#) を使用します。

インスタンス命名規則

XST では、インスタンス名が次の規則に基づいて作成されます（命名規則は、優先順にリストしています）。

1. インスタンス名で階層を維持します。
 - ・ 階層区切り文字は、[\[Hierarchy Separator\] \(-hierarchy_separator\)](#) で定義します。
 - ・ XST では、デフォルトの階層区切り文字はスラッシュ (/) です。
2. インスタンス名が HDL の generate 文で生成される場合、generate 文からのラベルがインスタンス名の一部に使用されます。
 - ・ たとえば、次のような VHDL の generate 文があるとして。

```
il_loop: for i in 1 to 10 generate
  inst_lut:LUT2 generic map (INIT => "00")
```

- ・ XST では、LUT 2 に対して次のインスタンス名が生成されます。

```
il_loop[1].inst_lut
il_loop[2].inst_lut
...
il_loop[9].inst_lut
il_loop[10].inst_lut
```

3. フリップフロップのインスタンス名をそれが駆動する信号名に合わせます。この原則は、ステートビットにも適用されます。
4. クロック バッファ インスタンス名には、出力信号名の後にアンダースコアとクロック バッファ タイプを付けます（例：_BUFGP や _IBUFG など）。
5. ブラック ボックスのインスタンス名は保持されます。
6. ライブラリ プリミティブのインスタンス名は保持されます。
7. 入力および出力バッファ名には、パッド名の後に _IBUF または _OBUF を付けます。
8. IBUF エLEMENTの出力インスタンスには instance_name_IBUF という形式の名前を付けます。
9. OBUF エLEMENTの入力インスタンスには instance_name_OBUF という形式の名前を付けます。

大文字/小文字の保持

- ・ このセクションでは、Verilog および VHDL での XST の大文字/小文字の保持について説明します。
- ・ 詳細は、「[大文字/小文字の区別](#)」を参照してください。

VHDL（大文字/小文字の区別なし）

- ・ VHDL では大文字と小文字は区別されません。
- ・ `case` コマンドライン オプション (`-case`) で指示されていない限り、HDL ソースコードで定義された名前に基づいたオブジェクト名が合成済みネットリストではすべて小文字に変換されて記述されます。

Verilog (大文字/小文字の区別あり)

- ・ Verilog では、大文字と小文字が区別されます。
- ・ [case](#) コマンドライン オプション (-case) で指示されていない限り、XST では HDL ソースコードと同じ大文字/小文字が使用されます。

命名規則の制御方法

- ・ 次の制約を使用すると、合成済みネットリストのオブジェクト名の表記をある程度制御できます。
 - [階層区切り文字](#)
 - [バス区切り文字](#)
 - [大文字/小文字](#)
 - [複製接尾語の設定](#)
- ・ これらの制約は次のいずれかの方法で適用します。
 - ISE® Design Suite
 - [Synthesize - XST] プロセス プロパティ
 - コマンド ライン
- ・ 詳細は、[第 9 章「デザイン制約」](#)を参照してください。

その他のリソース

- ・ ザイリンクス用語集 : <http://japan.xilinx.com/company/terms.htm>
- ・ ザイリンクス サポートおよび資料 : <http://japan.xilinx.com/support>

XST の詳細は、ザイリンクス サポート ページの「よくある質問 (FAQ)」を参照してください。

DSP ブロック リソースの詳細は、次を参照してください。

- ・ 『Virtex®-6 FPGA DSP48E1 スライス ユーザー ガイド』(UG369)
- ・ 『Spartan®-6 FPGA DSP48A1 スライス ユーザー ガイド』(UG389)
- ・ 『7 シリーズ DSP48E1 スライス ユーザー ガイド』(UG479)