

XST ユーザー ガイ ド (Virtex-6 および Spartan-6 デバイス用)

UG687 (v11.4) 2009 年 12 月 2 日

ザイリンクス商標および著作権情報



Xilinx is disclosing this user guide, manual, release note, and/or specification (the “Documentation”) to you solely for use in the development of designs to operate with Xilinx hardware devices. You may not reproduce, distribute, republish, download, display, post, or transmit the Documentation in any form or by any means including, but not limited to, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Xilinx. Xilinx expressly disclaims any liability arising out of your use of the Documentation. Xilinx reserves the right, at its sole discretion, to change the Documentation without notice at any time. Xilinx assumes no obligation to correct any errors contained in the Documentation, or to advise you of any corrections or updates. Xilinx expressly disclaims any liability in connection with technical support or assistance that may be provided to you in connection with the Information.

THE DOCUMENTATION IS DISCLOSED TO YOU “AS-IS” WITH NO WARRANTY OF ANY KIND. XILINX MAKES NO OTHER WARRANTIES, WHETHER EXPRESS, IMPLIED, OR STATUTORY, REGARDING THE DOCUMENTATION, INCLUDING ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT OF THIRD-PARTY RIGHTS. IN NO EVENT WILL XILINX BE LIABLE FOR ANY CONSEQUENTIAL, INDIRECT, EXEMPLARY, SPECIAL, OR INCIDENTAL DAMAGES, INCLUDING ANY LOSS OF DATA OR LOST PROFITS, ARISING FROM YOUR USE OF THE DOCUMENTATION.

© Copyright 2002–2009 Xilinx Inc. All Rights Reserved. XILINX, the Xilinx logo, the Brand Window and other designated brands included herein are trademarks of Xilinx, Inc. All other trademarks are the property of their respective owners. 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.11.4) を翻訳したもので、内容に相違が生じる場合には原文を優先します。
資料によっては英語版の更新に対応していないものがあります。
日本語版は参考用としてご使用の上、最新情報につきましては、必ず最新英語版をご参照ください。

目次

ザイリックス商標および著作権情報	2
1 : XST ユーザー ガイドについて	19
このマニュアルについて	19
マニュアルの内容	20
略語	20
その他のリソース	21
表記規則	21
書体	21
オンライン マニュアル	21
2 : XST の概要	23
Xilinx Synthesis Technology (XST) について	23
このリリースの新機能	24
3 : XST プロジェクトの作成および合成	25
HDL 合成プロジェクトの作成	25
HDL プロジェクト ファイルのコード例	26
ISE Design Suite での XST の実行	26
コマンド ライン モードからの XST の実行	26
XST をスタンドアロン ツールとして実行	27
XST をインタラクティブに実行	27
XST をスクリプト モードで実行	27
XST のスクリプト ファイル	27
XST のコマンド	28
run コマンド	28
set コマンド	29
script コマンド	29
help コマンド	29
サポートされるファミリ	30
特定デバイスの全コマンド	30
特定デバイスの特定コマンド	30
XST スクリプト ファイルの可読性の改善	31
XST の出力ファイル	31
XST の通常の出力ファイル	32
XST の一時出力ファイル	32
コマンド ライン モードでのスペースを含む名前	32
4 : VHDL 言語のサポート	33
VHDL の利点	34
VHDL の IEEE サポート	34
VHDL のデータ型	34
VHDL でサポートされるデータ型	34
定義済み列挙型	35
ユーザー定義の列挙型	35
ビット ベクタ型	36
整数型	36
多次元配列型	36
VHDL のレコード型	38
VHDL でサポートされないデータ型	38
VHDL のオブジェクト	38
VHDL の信号	38
VHDL の変数	38
VHDL の定数	39
VHDL の演算子	39
VHDL のエンティティとアーキテクチャの記述	40
VHDL の回路記述	40
VHDL のエンティティ宣言	40
VHDL のアーキテクチャ宣言	42

VHDL のコンポーネント インスタンス化	42
VHDL の再帰的なコンポーネント インスタンス化文	43
VHDL のコンポーネント コンフィギュレーション文	44
VHDL のジェネリック	45
VHDL のジェネリックと属性の競合	46
VHDL の組み合わせ回路	47
VHDL の同時処理信号代入文	47
VHDL のジェネレート文	49
VHDL の for-generate 文	49
VHDL の if-generate 文	50
VHDL の組み合わせプロセス文	51
VHDL の組み合わせプロセスの概要	51
VHDL の変数および信号の代入文	52
VHDL の if-else 文	53
VHDL の case 文	54
VHDL の for-loop 文	55
VHDL の順序ロジック	56
VHDL のセンシティビティリスト付き順次プロセス文	56
VHDL のセンシティビティリストのない順次プロセス文	57
VHDL の初期値と動作に合わせたセット/リセット	58
VHDL のメモリ エLEMENTのデフォルト初期値	59
VHDL の関数とプロシージャ	60
VHDL のアサート文	64
VHDL ライブラリおよびパッケージ	66
VHDL ライブラリ	66
VHDL の定義済みパッケージ	66
VHDL の定義済み標準パッケージ	67
VHDL の定義済み IEEE パッケージ	67
VHDL の定義済み IEEE 実数型および IEEE math_real パッケージ	67
独自の VHDL パッケージの定義	68
VHDL パッケージのアクセス	69
VHDL のファイル タイプ サポート	69
XST での VHDL ファイルの読み出し/書き込み機能	70
外部ファイルからのメモリ内容の読み込み	72
デバッグ用ファイルへの書き込みコード例	72
書き込み関数を使用したデバッグの規則	75
VHDL 構文	75
VHDL のデザイン エンティティとコンフィギュレーション	75
VHDL の論理式	76
VHDL 演算子のサポート	77
VHDL オペランドのサポート	77
VHDL 文	77
VHDL の予約語	79
5 : Verilog のサポート	81
Verilog のサポート	81
変数による部分的ビット選択	82
部分的なビット選択の Verilog コード例	82
Verilog 構造記述	83
Verilog パラメータ	84
Verilog パラメータと属性の競合	86
Verilogの使用制限	87
大文字/小文字の区別	87
ブロッキングおよびノンブロッキング代入文	87
整数処理	88
case 文での整数処理	88
連結文での整数処理	89
Verilog -2001 の属性とメタ コメント	89
Verilog-2001 の属性	89
Verilog のメタ コメント	89

Verilog 構文	90
Verilog の定数	91
Verilog のデータ型	91
Verilog の継続代入文	91
Verilog の手続き代入文	91
Verilog のデザイン階層	92
Verilog コンパイラ指示子	93
Verilog のタスクおよび関数	93
サポートされるシステム タスクと関数	94
変換関数の使用	94
ファイル I/O タスクを使用したメモリ内容の読み込み	94
ディスプレイタスク	95
\$finish を使用したデザイン ルール チェックの作成	95
Verilog プリミティブ	97
Verilog の予約言語	98
Verilog2001 のサポート	100
6 : Verilog ビヘイビア記述のサポート	101
Verilog ビヘイビア記述の変数宣言	101
Verilog ビヘイビア記述の初期値	102
Verilog ビヘイビア記述の配列のコード例	102
Verilog ビヘイビア記述の多次元配列	103
Verilog ビヘイビア記述のデータ型	103
Verilog ビヘイビア記述で使用可能な文	104
Verilog ビヘイビア記述の論理式	104
Verilog ビヘイビア記述の論理式の概要	104
Verilog ビヘイビア言語でサポートされる演算子	105
Verilog ビヘイビア記述でサポートされる論理式	105
Verilog のビヘイビア記述の論理式の評価結果	107
Verilog ビヘイビア記述のブロック	107
Verilog ビヘイビア記述のモジュール	108
Verilog ビヘイビア記述のモジュール宣言	108
Verilog ビヘイビア記述のモジュール インスタンス化	108
Verilog ビヘイビア記述の継続代入文	109
Verilog ビヘイビア記述の手続き代入文	109
Verilog ビヘイビア記述の手続き代入文の概要	110
組み合わせ always ブロック	110
if-else 文	110
case 文	111
for および repeat ループ文	112
while 文	113
順次 always ブロック	114
assign 文および deassign 文	116
32 ビットを超える場合のビットの拡張	116
Verilog ビヘイビア記述のタスクおよび関数	116
Verilog ビヘイビア記述のタスクおよび関数	116
Verilog ビヘイビア記述のタスクおよび関数のコード例	117
Verilog ビヘイビア記述の再帰タスクおよび関数	118
Verilog ビヘイビア記述の定数関数	119
Verilog ビヘイビア記述のブロックおよびノンブロッキング手続き代入文	119
Verilog ビヘイビア記述の定数	120
Verilog ビヘイビア記述のマクロ	120
Verilog ビヘイビア記述の include ファイル	121
Verilog ビヘイビア記述のコメント	121
Verilog ビヘイビア記述の generate 文	122
Verilog ビヘイビア記述の generate 文	122
Verilog ビヘイビア記述の generate ループ文	122
Verilog ビヘイビア記述の generate 条件文	123
Verilog ビヘイビア記述の generate-case 文	123
7 : 混合言語のサポート	125

混合言語サポートの概要	126
VHDL/Verilog の境界規則	126
VHDL/Verilog の境界規則	126
Verilog デザインへの VHDL デザイン ユニットのインスタンスシート	127
VHDL への Verilog モジュールのインスタンスシート	127
ポートのマッピング	128
Verilog への VHDL のインスタンスシート	128
VHDL への Verilog のインスタンスシート	129
ジェネリックのサポート	129
LSO ファイル	129
LSO ファイルの概要	129
ISE Design Suite での LSO ファイルの指定	130
コマンドライン モードでの LSO ファイルの指定	130
LSO の規則	130
空の LSO ファイル	130
DEFAULT_SEARCH_ORDER キーワードのみの場合	130
DEFAULT_SEARCH_ORDER キーワードとライブラリリストがある場合	131
ライブラリリストのみの場合	131
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	132
8 : HDL コーディング手法	133
HDL コーディング手法の概要	133
記述言語の選択	134
マクロ推論フローの概要	134
フリップフロップおよびレジスタ	135
フリップフロップおよびレジスタの概要	135
フリップフロップおよびレジスタの初期化	135
フリップフロップおよびレジスタの制御信号	136
フリップフロップおよびレジスタの関連制約	136
フリップフロップおよびレジスタのレポート	137
フリップフロップおよびレジスタのコード例	137
ラッチ	139
ラッチの概要	139
ラッチの記述	140
ラッチの関連制約	140
ラッチのレポート	141
ラッチのコード例	141
トライステート	143
トライステートの概要	143
トライステートのインプリメンテーション	143
トライステートの関連制約	143
トライステートのレポート	144
トライステートのコード例	144
カウンタおよびアキュムレータ	148
カウンタおよびアキュムレータ	149
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	149
カウンタおよびアキュムレータの関連制約	150
カウンタおよびアキュムレータのレポート	151
カウンタおよびアキュムレータのコード例	152
シフトレジスタ	153
シフトレジスタの概要	153
シフトレジスタの記述	154
シフトレジスタのインプリメンテーション	154
シフトレジスタの SRL ベースのインプリメンテーション	154
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	155
LUT RAM へのシフトレジスタのインプリメンテーション	157
シフトレジスタの関連制約	159
シフトレジスタのレポート	159
シフトレジスタのコード例	159
ダイナミック シフトレジスタ	163

ダイナミック シフトレジスタの概要	163
ダイナミック シフトレジスタの関連制約	164
ダイナミック シフトレジスタのレポート	164
ダイナミック シフトレジスタのコード例	164
マルチプレクサ	166
マルチプレクサの概要	166
マルチプレクサのインプリメンテーション	167
マルチプレクサの Verilog の [Case Implementation Style] パラメータ	167
マルチプレクサの関連制約	167
マルチプレクサのレポート	168
マルチプレクサのコード例	168
四則演算	173
四則演算ブロックの概要	173
XST での四則演算の符号サポート	173
XST での四則演算の符号サポート	174
Verilog の符号サポート	174
VHDL の符号サポート	175
四則演算のインプリメンテーション	175
四則演算のスライス ロジック	175
四則演算の DSP ブロックリソース	175
コンパレータ	176
コンパレータの概要	176
コンパレータの関連制約	177
コンパレータのレポート	177
コンパレータのコード例	177
除算器	179
除算器の概要	179
除算器の関連制約	179
除算器のレポート	179
除算器のコード例	179
加算器、減算器、加減算器	181
加算器、減算器、加減算器の概要	181
キャリー出力の記述	181
加算器、減算器、加減算器のインプリメンテーション	182
加算器、減算器、加減算器の関連制約	182
加算器、減算器、加減算器のレポート	183
加算器、減算器、加減算器のコード例	183
乗算器	185
乗算器の概要	185
乗算器のインプリメンテーション	185
乗算器のインプリメンテーションの概要	185
DSP ブロックのインプリメンテーション	186
スライス ロジックのインプリメンテーション	186
定数への乗算	186
乗算器の関連制約	187
乗算器のレポート	187
乗算器のコード例	188
乗加算および乗累算	190
乗加算および乗累算	191
乗加算と乗累算のインプリメンテーション	191
乗加算と乗累算の関連制約	191
乗加算と乗累算のレポート	191
乗加算と乗累算のコード例	192
DSP の推論	194
DSP の推論の概要	195
対称フィルタ	195
DSP の推論のコード例	195
リソース共有	196
リソース共有の概要	196

リソース共有の関連制約	196
リソース共有のレポート	197
リソース共有のコード例	197
RAM	199
RAM の概要	199
分散 RAM とブロック RAM	199
RAM のサポート機能	200
RAM の HDL コード記述のガイドライン	201
RAM の記述	201
書き込みアクセスの記述	201
VHDL での書き込みアクセスの記述	202
Verilog での書き込みアクセスの記述	202
読み出しアクセスの記述	202
VHDL での読み出しアクセスの記述	203
Verilog での読み出しアクセスの記述	203
ブロック RAM の読み出し/書き込みの同期	203
リセット可能なデータ出力 (ブロック RAM)	205
バイト幅の書き込みイネーブルのサポート (ブロック RAM)	206
非対称ポートのサポート	207
RAM の初期内容	208
HDL ソースコードでの初期内容の指定	208
外部データ ファイルでの初期内容の指定	209
ブロック RAM の最適化ストラテジ	210
ブロック RAM の最適化ストラテジの概要	211
ブロック RAM のパフォーマンス	211
ブロック RAM のデバイス使用	211
ブロック RAM の電力	211
小型の RAM の条件	211
ブロック RAM へのロジックと FSM のマップ	212
ブロック RAM リソースの管理	212
ブロック RAM のパッキング	212
分散 RAM のパイプライン	212
RAM の関連制約	213
RAM のレポート	213
RAM のコード例	214
ROM	242
ROM の概要	242
ROM の詳細	243
ROM のコード記述	243
読み出しアクセスの記述	245
ROM のインプリメンテーション	246
ROM の関連制約	246
ROM のレポート	246
ROM のコード例	247
FSM コンポーネント	250
FSM コンポーネントの概要	250
FSM の説明	250
Finite State Machine (FSM) の概要	250
ステートレジスタ	251
次ステートの論理式	252
達成不可能ステート	252
FSM の出力	252
FSM の入力	252
ステート エンコード手法	252
自動ステート エンコード	252
ワンホット ステート エンコード	252
グレイ ステート エンコード	253
コンパクト ステート エンコード	253
ジョンソン ステート エンコード	253

シーケンシャル ステート エンコード	253
Speed1 ステート エンコード	253
ユーザー ステート エンコード	253
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	253
FSM セーフ インプリメンテーション	254
FSM 関連の制約	254
FSM のレポート	254
FSM のコード例	255
ブラック ボックス	258
ブラック ボックスの概要	258
ブラック ボックスの関連制約	258
ブラック ボックスのレポート	258
ブラック ボックスのコード例	259
9 : FPGA の最適化	261
下位レベルの合成	261
ブロック RAM へのロジックのマップ	262
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	262
フリップフロップのリタイミング	263
フリップフロップのリタイミングの概要	263
フリップフロップのリタイミングの制限	264
フリップフロップのリタイミングの制御方法	264
パーティション	264
エリア制約を設定した場合のスピード最適化	264
インプリメンテーション制約	265
ザイリンクス デバイス プリミティブのサポート	266
デバイス プリミティブのサポートの概要	266
属性を使用したプリミティブの生成	266
プリミティブとブラック ボックス	266
VHDL および Verilog のザイリンクス デバイス プリミティブ ライブラリ	267
VHDL および Verilog のザイリンクス デバイス プリミティブ ライブラリの概要	267
VHDL および Verilog のザイリンクス デバイス プリミティブ ライブラリの概要	267
VHDL のザイリンクス デバイス プリミティブ ライブラリ	267
Verilog のザイリンクス デバイス プリミティブ ライブラリ	267
プリミティブ インスタンス化のガイドライン	268
プリミティブ プロパティの指定	268
インスタンス化されたデバイス プリミティブのレポート	268
プリミティブの関連制約	269
プリミティブのコード例	269
UniMacro ライブラリの使用	271
コアの処理	271
コアの読み込み	271
コアの検索	272
コアのレポート	272
LUT へのロジックのマップ	272
デバイス上の配置の制御	274
バッファの挿入	275
XST での PCI フローの使用	275
XST での PCI フローの使用の概要	275
ロジックとフリップフロップの複製の回避	275
コアの自動読み込み機能のオフ	276
10 : デザイン制約	277
制約の概要	277
制約の指定	278
制約の優先順序	278
ISE Design Suite の合成オプション	278
ISE Design Suite での XST オプションの設定	279
その他の XST コマンドライン オプションの設定	279
デザイン目標とストラテジ	279
VHDL 属性	279

Verilog-2001 の属性.....	280
Verilog-2001 属性の概要.....	280
Verilog-2001 の構文.....	281
Verilog-2001 の制限.....	283
Verilog のメタ コメント.....	283
XST Constraint File (XCF).....	283
XCF の概要.....	283
ネイティブ UCF 制約とそれ以外の UCF 構文.....	285
ネイティブ UCF 制約.....	285
ネイティブ以外の UCF 制約.....	285
構文の制限.....	286
XCF ファイルからのみ適用可能なタイミング制約.....	286
11 : 一般制約.....	287
I/O バッファの追加 (-iobuf).....	288
アーキテクチャ サポート.....	289
適用可能エレメント.....	289
適用ルール.....	289
構文例.....	289
ボックス タイプ (BOX_TYPE).....	289
アーキテクチャ サポート.....	290
適用可能エレメント.....	290
適用ルール.....	290
構文例.....	290
バスの区切り文字指定 (-bus_delimiter).....	291
アーキテクチャ サポート.....	291
適用可能エレメント.....	291
適用ルール.....	291
構文例.....	291
大文字/小文字の指定 (-case).....	291
アーキテクチャ サポート.....	291
適用可能エレメント.....	291
適用ルール.....	292
構文例.....	292
Case 文のインプリメンテーション形式 (-vlgcase).....	292
アーキテクチャ サポート.....	293
適用可能エレメント.....	293
適用ルール.....	293
構文例.....	293
Verilog マクロ (-define).....	293
アーキテクチャ サポート.....	293
適用可能エレメント.....	293
適用ルール.....	294
構文例.....	294
複製接尾語の設定 (-duplication_suffix).....	294
アーキテクチャ サポート.....	295
適用可能エレメント.....	295
適用ルール.....	295
構文例.....	295
フル ケース (FULL_CASE).....	295
アーキテクチャ サポート.....	295
適用可能エレメント.....	296
適用ルール.....	296
構文例.....	296
RTL 回路図の生成 (-rtlview).....	297
アーキテクチャ サポート.....	297
適用可能エレメント.....	297
適用ルール.....	297
構文例.....	297
ジェネリック (-generics).....	297

アーキテクチャ サポート	298
適用可能エレメント	298
適用ルール	298
構文例	298
階層 区切り文字の指定 (-hierarchy_separator)	299
アーキテクチャ サポート	299
適用可能エレメント	299
適用ルール	299
構文例	300
I/O 規格 (IOSTANDARD)	300
キープ (KEEP)	300
階層の維持 (KEEP_HIERARCHY)	301
KEEP_HIERARCHY の値	301
階層の保持	301
階層維持の図	301
アーキテクチャ サポート	302
適用可能エレメント	302
適用ルール	302
構文例	302
ライブラリの検索順 (-lso)	303
アーキテクチャ サポート	303
適用可能エレメント	303
適用ルール	303
構文例	303
LOC	304
ネットリスト階層 (-netlist_hierarchy)	304
アーキテクチャ サポート	304
適用可能エレメント	304
適用ルール	304
構文例	304
最適化エフォート レベル (OPT_LEVEL)	305
アーキテクチャ サポート	305
適用可能エレメント	305
適用ルール	305
構文例	305
最適化方法の指定 (OPT_MODE)	306
アーキテクチャ サポート	306
適用可能エレメント	306
適用ルール	306
構文例	306
パラレル ケース (PARALLEL_CASE)	307
アーキテクチャ サポート	307
適用可能エレメント	307
適用ルール	307
構文例	307
RLOC	308
保存 (S / SAVE)	308
合成制約ファイルの指定 (-uc)	308
アーキテクチャ サポート	308
適用可能エレメント	309
適用ルール	309
構文例	309
変換なし (TRANSLATE_OFF) と変換あり (TRANSLATE_ON)	309
アーキテクチャ サポート	309
適用可能エレメント	309
適用ルール	309
構文例	310
合成制約ファイルの無視 (-iuc)	310
アーキテクチャ サポート	310

適用可能エレメント	310
適用ルール	310
構文例	310
Verilog の 'include ディレクトリの指定 (-vlgincdir)	311
アーキテクチャ サポート	311
適用可能エレメント	311
適用ルール	311
構文例	311
HDL ライブラリ マップ ファイル (-xsthdpini)	312
アーキテクチャ サポート	312
適用可能エレメント	313
適用ルール	313
構文例	313
作業ディレクトリの指定 (-xsthdpdir)	313
具体例	313
アーキテクチャ サポート	314
適用可能エレメント	314
適用ルール	314
構文例	314
12 : HDL 制約	315
FSM 自動抽出 (FSM_EXTRACT)	315
アーキテクチャ サポート	315
適用可能エレメント	315
適用ルール	315
構文例	315
列挙型エンコード手法 (ENUM_ENCODING)	316
アーキテクチャ サポート	316
適用可能エレメント	317
適用ルール	317
構文例	317
等価レジスタの削除 (EQUIVALENT_REGISTER_REMOVAL)	317
アーキテクチャ サポート	318
適用可能エレメント	318
適用ルール	318
構文例	318
FSM エンコード方法の指定 (FSM_ENCODING)	319
アーキテクチャ サポート	319
適用可能エレメント	319
適用ルール	319
構文例	319
MUX の抽出 (MUX_EXTRACT)	320
アーキテクチャ サポート	320
適用可能エレメント	321
適用ルール	321
構文例	321
MUX の最小サイズ (MUX_MIN_SIZE)	322
アーキテクチャ サポート	322
適用可能エレメント	322
適用ルール	322
構文例	322
リソース共有 (RESOURCE_SHARING)	323
アーキテクチャ サポート	323
適用可能エレメント	323
適用ルール	323
構文例	323
セーフリカバリ ステート (SAFE_RECOVERY_STATE)	324
アーキテクチャ サポート	324
適用可能エレメント	324
適用ルール	324

構文例.....	325
セーフ インプリメンテーション (SAFE_IMPLEMENTATION)	325
アーキテクチャ サポート	325
適用可能エレメント	325
適用ルール.....	325
構文例.....	326
13 : FPGA 制約 (タイミング制約以外).....	327
非同期から同期への変換 (ASYNC_TO_SYNC).....	328
アーキテクチャ サポート	329
適用可能エレメント	329
適用ルール.....	329
構文例.....	329
自動 BRAM パッキング (AUTO_BRAM_PACKING).....	329
アーキテクチャ サポート	329
適用可能エレメント	329
適用ルール.....	330
構文例.....	330
BRAM 使用率 (BRAM_UTILIZATION_RATIO).....	330
アーキテクチャ サポート	330
適用可能エレメント	330
適用ルール.....	331
構文例.....	331
バッファタイプ (BUFFER_TYPE)	332
アーキテクチャ サポート	332
適用可能エレメント	332
適用ルール.....	332
構文例.....	332
BUFGCE の抽出 (BUFGCE)	332
アーキテクチャ サポート	333
適用可能エレメント	333
適用ルール.....	333
構文例.....	333
コアの検索ディレクトリ (-sd)	333
アーキテクチャ サポート	333
適用可能エレメント	334
適用ルール.....	334
構文例.....	334
DSP 使用率 (DSP_UTILIZATION_RATIO).....	334
アーキテクチャ サポート	334
適用可能エレメント	334
適用ルール.....	335
構文例.....	335
FSM スタイル (FSM_STYLE)	335
アーキテクチャ サポート	335
適用可能エレメント	336
適用ルール.....	336
構文例.....	336
電力削減 (POWER)	337
アーキテクチャ サポート	337
適用可能エレメント	337
適用ルール.....	337
構文例.....	337
コアの読み込み (READ_CORES).....	338
アーキテクチャ サポート	338
適用可能エレメント	339
適用ルール.....	339
構文例.....	339
再合成 (RESYNTHESIZE)	340
アーキテクチャ サポート	340

適用可能エレメント	340
適用ルール	340
構文例	340
インクリメンタル合成 (INCREMENTAL_SYNTHESIS)	341
アーキテクチャ サポート	341
適用可能エレメント	341
適用ルール	341
構文例	341
LUT の結合 (LC)	342
アーキテクチャ サポート	342
適用可能エレメント	342
適用ルール	342
構文例	342
BRAM へのロジックのマッピング (BRAM_MAP)	342
アーキテクチャ サポート	343
適用可能エレメント	343
適用ルール	343
構文例	343
最大ファンアウト数 (MAX_FANOUT)	343
アーキテクチャ サポート	344
適用可能エレメント	344
適用ルール	344
構文例	345
最初のフリップフロップ ステージの移動 (MOVE_FIRST_STAGE)	346
アーキテクチャ サポート	346
適用可能エレメント	347
適用ルール	347
構文例	347
最後のフリップフロップ ステージの移動 (MOVE_LAST_STAGE)	348
アーキテクチャ サポート	348
適用可能エレメント	348
適用ルール	348
構文例	348
乗算器スタイル (MULT_STYLE)	349
アーキテクチャ サポート	349
適用可能エレメント	349
適用ルール	349
構文例	349
グローバル クロック バッファ数 (-bufg)	350
アーキテクチャ サポート	350
適用可能エレメント	350
適用ルール	350
構文例	350
インスタンス化されたプリミティブの最適化 (OPTIMIZE_PRIMITIVES)	351
アーキテクチャ サポート	351
適用可能エレメント	351
適用ルール	351
構文例	351
I/O レジスタの IOB 内へのパック (IOB)	352
RAM の抽出 (RAM_EXTRACT)	352
アーキテクチャ サポート	353
適用可能エレメント	353
適用ルール	353
構文例	353
RAM スタイル (RAM_STYLE)	354
アーキテクチャ サポート	354
適用可能エレメント	354
適用ルール	354
構文例	354

制御セットの削減 (REDUCE_CONTROL_SETS)	355
アーキテクチャ サポート	356
適用可能エレメント	356
適用ルール	356
構文例	356
レジスタの自動調整 (REGISTER_BALANCING)	356
順方向のレジスタ自動調整	356
逆方向のレジスタ自動調整	357
レジスタ自動調整の値	358
レジスタ自動調整に影響するその他の制約	358
アーキテクチャ サポート	359
適用可能エレメント	359
適用ルール	359
構文例	359
レジスタの複製 (REGISTER_DUPLICATION)	360
アーキテクチャ サポート	360
適用可能エレメント	360
適用ルール	360
構文例	360
ROM の抽出 (ROM_EXTRACT)	361
アーキテクチャ サポート	361
適用可能エレメント	362
適用ルール	362
構文例	362
ROM スタイル (ROM_STYLE)	362
アーキテクチャ サポート	363
適用可能エレメント	363
適用ルール	363
構文例	363
シフトレジスタの抽出 (SHREG_EXTRACT)	364
アーキテクチャ サポート	364
適用可能エレメント	364
適用ルール	364
構文例	365
スライス パッキング (-slice_packing)	365
アーキテクチャ サポート	365
適用可能エレメント	366
適用ルール	366
構文例	366
ロー スキュー ラインの使用 (USE_LOWSKEW_LINES)	366
Slice (LUT-FF Pairs) Utilization Ratio (スライス (LUT-FF ペア) 使用率)	366
アーキテクチャ サポート	366
適用可能エレメント	366
適用ルール	366
構文例	367
SLICE_UTILIZATION_RATIO_MAXMARGIN (スライス (LUT-FF ペア) 使用率の許容範囲)	368
アーキテクチャ サポート	368
適用可能エレメント	368
適用ルール	368
構文例	368
単一 LUT へのエンティティのマッピング (LUT_MAP)	369
アーキテクチャ サポート	370
適用可能エレメント	370
適用ルール	370
構文例	370
キャリーチェーンの使用 (USE_CARRY_CHAIN)	370
アーキテクチャ サポート	371
適用可能エレメント	371
適用ルール	371

構文例.....	371
トライステートからロジックへの変換 (TRISTATE2LOGIC).....	372
制限事項	372
アーキテクチャ サポート	372
適用可能エレメント	372
適用ルール.....	373
構文例.....	373
クロック イネーブルの使用 (USE_CLOCK_ENABLE).....	373
アーキテクチャ サポート	374
適用可能エレメント	374
適用ルール.....	374
構文例.....	374
同期セットの使用 (USE_SYNC_SET).....	375
アーキテクチャ サポート	375
適用可能エレメント	376
適用ルール.....	376
構文例.....	376
同期リセットの使用 (USE_SYNC_RESET).....	377
アーキテクチャ サポート	377
適用可能エレメント	377
適用ルール.....	377
構文例.....	377
DSP ブロックの使用 (USE_DSP48)	378
アーキテクチャ サポート	379
適用可能エレメント	379
適用ルール.....	379
構文例.....	379
14 : タイミング制約	381
タイミング制約の指定	381
タイミング制約の指定の概要	381
グローバル最適化オプションを使用したタイミング制約の指定	382
UCF を使用したタイミング制約の指定	382
NGC ファイルへの制約の書き込み	382
タイミング制約の処理に影響のあるその他のオプション	382
クロス クロック解析 (-cross_clock_analysis)	382
アーキテクチャ サポート	382
適用可能エレメント	383
適用ルール.....	383
構文例.....	383
タイミング制約の書き込み (-write_timing_constraints)	383
アーキテクチャ サポート	383
適用可能エレメント	383
適用ルール.....	383
構文例.....	383
クロック信号 (CLOCK_SIGNAL)	384
アーキテクチャ サポート	384
適用可能エレメント	384
適用ルール.....	384
構文例.....	384
グローバルな最適化目標 (-glob_opt).....	385
グローバル最適化のドメインの定義	386
XCF のタイミング制約のサポート	386
周期 (PERIOD).....	387
アーキテクチャ サポート	387
適用可能エレメント	387
適用ルール.....	387
構文例.....	387
オフセット (OFFSET).....	387
アーキテクチャ サポート	388

適用可能エレメント	388
適用ルール	388
構文例	388
From-To (FROM-TO)	388
アーキテクチャ サポート	388
適用可能エレメント	388
適用ルール	388
構文例	388
タイミング名 (TNM)	389
アーキテクチャ サポート	389
適用可能エレメント	389
適用ルール	389
構文例	389
ネットのタイミング名 (TNM_NET)	389
アーキテクチャ サポート	389
適用可能エレメント	389
適用ルール	389
構文例	390
タイムグループ (TIMEGRP)	390
アーキテクチャ サポート	390
適用可能エレメント	390
適用ルール	390
構文例	390
タイミング無視 (TIG)	390
アーキテクチャ サポート	390
適用可能エレメント	391
適用ルール	391
構文例	391
15 : サードパーティの制約	393
サードパーティの制約と同等の XST 制約	393
サードパーティ制約の構文例	397
Verilog 構文例	397
XCF の構文例	397
16 : XST 合成レポート	399
XST 合成レポートの概要	399
XST 合成レポートの内容	400
XST 合成レポートの目次	400
合成オプションのサマリ - Synthesis Options Summary セクション	400
XST 合成レポート - HDL Parsing および Elaboration セクション	400
XST 合成レポート - HDL Synthesis セクション	400
XST 合成レポート - Advanced HDL Synthesis セクション	401
XST 合成レポート - Low Level Synthesis セクション	401
XST 合成レポート - Partition Report セクション	401
XST 合成レポート - Design Summary セクション	401
プリミティブおよびブラック ボックスの使用率	401
デバイス使用率のサマリ	402
パーティション リソース サマリ	402
タイミング レポート	402
クロック情報	402
非同期制御信号の情報	403
タイミング サマリ	403
タイミングの詳細	404
暗号化されたモジュール	404
XST 合成レポートのナビゲーション	405
コマンド ライン モードのレポート ナビゲーション	405
ISE Design Suite のレポート ナビゲーション	405
XST 合成レポートの情報	405
メッセージ フィルタ	405
Quiet モード	406

Silent モード	406
17 : XST の命名規則	407
命名規則の概要	407
XST 命名規則のコード例	407
always ブロック (ラベルあり) の reg を示す Verilog コード例	408
if-generate 文 (ラベルなし) でプリミティブ インスタンス化を記述した Verilog コード例	409
if-generate 文 (ラベルあり) でプリミティブ インスタンス化を記述した Verilog コード例	410
process 文 (ラベルあり) の変数を示す VHDL コード例	411
ブール型で記述したフリップフロップの VHDL コード例	412
ネットの命名規則	413
インスタンスの命名規則	414
大文字/小文字の保持	414
命名規則の制御方法	414

第 1 章

XST ユーザー ガイドについて

XST ユーザー ガイドには、次の内容が含まれます。

- ・ XST (Xilinx® Synthesis Technology) における Hardware Description Language (HDL) 言語、ザイリンクス デバイス、およびザイリンクスの ISE® Design Suite ソフトウェアで使用されるデザイン制約のサポート
- ・ XST を使用してデザインを作成する際の FPGA および CPLD 最適化の手法
- ・ ISE Design Suite の [Processes] ウィンドウおよびコマンド ラインからの XST 実行方法

このマニュアルについて

メモ: 『XST ユーザー ガイド (Virtex-6 および Spartan-6 用)』は、Virtex®-6 および Spartan®-6 デバイス専用のマニュアルです。その他のデバイスを使用して XST を実行する場合は、『XST ユーザー ガイド』を参照してください。

『XST ユーザー ガイド (Virtex®-6 および Spartan®-6 デバイス用)』は、リファレンスおよび具体的な方法のガイドとして使用できます。含まれる内容は、次のとおりです。

- ・ XST (Xilinx Synthesis Technology) 合成ツールの実行および制御に関する詳細
- ・ HDL (ハードウェア記述言語) を使用して回路を設計する際のコーディング手法
- ・ ビルトインの最適化手法、最適なインプリメンテーションの達成に関するガイドライン

次のセクションが含まれています。

- ・ マニュアルの内容
- ・ 略語
- ・ その他のリソース
- ・ 表記規則

マニュアルの内容

『XST ユーザー ガイド (Virtex-6 および Spartan-6 用)』には、次が含まれています。

- ・ 第 1 章「XST の概要」: Xilinx Synthesis Technology (XST) 合成ツールの概略について説明します。
- ・ 第 2 章「XST プロジェクトの作成および合成」: XST を開始する際に役立つ情報が含まれ、HDL 合成プロジェクトの作成方法や XST を制御および実行する方法について説明します。
- ・ 第 3 章「VHDL 言語のサポート」: XST での VHSIC Hardware Description Language (VHDL) サポート、サポートされている構文、合成オプションについて詳しく説明します。
- ・ 第 4 章「Verilog 言語のサポート」: XST でサポートされている Verilog 構文およびメタ コメントなどについて説明します。
- ・ 第 5 章「Verilog ビヘイビア記述のサポート」: XST でサポートされている Verilog のビヘイビア記述について説明します。
- ・ 第 6 章「混合言語のサポート」: Verilog/VHDL デザインの混ざった XST プロジェクトの実行方法について説明します。
- ・ 第 7 章「HDL コーディング手法」: デジタル ロジック回路のコード例が含まれます。
- ・ 第 8 章「FPGA の最適化」: FPGA を最適化するための制約の使用方法、マクロ生成、FPGA デバイスでサポートされているプリミティブについて説明します。
- ・ 第 9 章「デザイン制約」: XST のデザイン制約に関する一般的な情報を提供します。
- ・ 第 10 章「一般制約」: XST の一般制約について個別に説明します。
- ・ 第 11 章「HDL 制約」: XST の HDL 制約について個別に説明します。
- ・ 第 12 章「FPGA 制約 (タイミング制約以外)」: XST の FPGA 制約 (タイミング制約以外) について個別に説明します。
- ・ 第 13 章「タイミング制約」: XST のタイミング制約について説明します。
- ・ 第 14 章「サポートされるサードパーティ制約」: XST でサポートされるサードパーティ制約について説明します。
- ・ 第 15 章「合成レポート」: XST のログ ファイルについて説明します。
- ・ 第 16 章「命名規則」: XST の命名規則について説明します。

略語

短縮形	意味
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	有限状態マシン (FSM)
EDIF	Electronic Data Interchange Format (電子データ交換フォーマット)
LSO	Library Search Order (ライブラリ検索順)
XST	Xilinx® Synthesis Technology の略。
XCF	XST Constraint File (XST 制約ファイル)

その他のリソース

その他の資料については、次の Web サイトから参照してください。

<http://japan.xilinx.com/literature>

シリコン、ソフトウェア、IP に関する質問および解答をアンサー データベースで検索したり、テクニカル サポートのウェブ ケースを開くには、次のザイリンクス Web サイトにアクセスしてください。

<http://japan.xilinx.com/support>

表記規則

このマニュアルでは、次の表記規則を使用しています。各規則について、例を挙げて説明します。

書体

次の規則は、すべてのマニュアルで使用されています。

表記規則	使用箇所	例
Courier フォント	システムが表示するメッセージ、プロンプト、プログラム ファイルを表示します。	<code>speed grade: - 100</code>
Courier フォント (太字)	構文内で入力するコマンドを示します。	ngdbuild <i>design_name</i>
イタリック フォント	ユーザーが値を入力する必要のある構文内の変数に使用します。	<i>ngdbuild design_name</i>
二重/一重かぎカッコ『』、『』、「」	『』はマニュアル名を、「」はセクション名を示します。	詳細については、『コマンドライン ツール ユーザー ガイド』の「PAR」を参照してください。
角カッコ []	オプションの入力またはパラメータを示しますが、 bus[7:0] のようなバス仕様では必ず使用します。また、GUI 表記にも使用します。	<code>ngdbuild [option_name] design_name</code> [File] → [Open] をクリックします。
中カッコ { }	1 つ以上の項目を選択するためのリストを示します。	lowpwr = {on off}
縦棒	選択するリストの項目を分離します。	lowpwr = {on off}
縦の省略記号	繰り返し項目が省略されていることを示します。	IOB #1: Name = QOUT IOB #2: Name = CLKIN . . .
横の省略記号 . . .	繰り返し項目が省略されていることを示します。	allow block . . . block_name <i>loc1 loc2 ... locn;</i>

オンライン マニュアル

このマニュアルでは、次の規則が使用されています。

表記規則	使用箇所	例
青色の文字	マニュアル内の相互参照、その他の文書へのリンクを示します。	詳細については、「 その他のリソース 」を参照してください。 詳細については、第 1 章「 タイトル フォーマット 」を参照してください。 詳細は、『 Virtex-6 ハンドブック 』の図 25 を参照してください。

XST の概要

メモ: 『XST ユーザー ガイド (Virtex-6 および Spartan-6 用)』は、Virtex®-6 および Spartan®-6 デバイス専用のマニュアルです。その他のデバイスを使用して XST を実行する場合は、『XST ユーザー ガイド』を参照してください。

この章では、Xilinx Synthesis Technology (XST) の一般的な情報と今回のリリースでの変更点について説明します。次のセクションが含まれています。

- ・ [Xilinx Synthesis Technology \(XST\) について](#)
- ・ [このリリースの新機能](#)

Xilinx Synthesis Technology (XST) について

Xilinx Synthesis Technology (XST) ソフトウェア

- ・ 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

Xilinx Synthesis Technology (XST) ソフトウェア

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

デザインの論理記述である合成済みネットリストは

1. 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。
2. 物理記述に変換されます。
3. 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。

XST の詳細は、ザイリンクス サポート ページの「Xilinx Synthesis Technology (XST) - よくある質問 (FAQ)」を参照してください。XST FAQ というキーワードで検索してください。

このリリースの新機能

このリリースからは、次の機能が追加されています。

- ・ 次のデバイスがサポートされるようになりました。
 - Spartan®-6 XA オートモーティブ デバイス ファミリ
 - Spartan-6 -4 スピードグレード
 - Spartan-6 -1L スピードグレード
- ・ 新しい制約 [MUX_MIN_SIZE](#)
この制約は、Virtex®-6 および Spartan-6 デザインのデバイス使用率を改善するためのものです。
- ・ **\$finish** システム タスクを使用した Verilog でのデザイン ルール チェック
詳細は、「[\\$finish を使用したデザイン ルール チェックの作成](#)」を参照してください。

XST プロジェクトの作成および合成

メモ: 『XST ユーザー ガイド (Virtex-6 および Spartan-6 用)』は、Virtex®-6 および Spartan®-6 デバイス専用のマニュアルです。その他のデバイスを使用して XST を実行する場合は、『XST ユーザー ガイド』を参照してください。

この章では、XST プロジェクトの作成と合成について次のセクションに分けて説明します。

- ・ [HDL 合成プロジェクトの作成](#)
- ・ [ISE® Design Suite での XST の実行](#)
- ・ [コマンド ライン モードからの XST の実行](#)
- ・ [XST の出力ファイル](#)

HDL 合成プロジェクトの作成

XST はほかの合成ツールとは異なり、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。

- ・ デザインに関する情報は HDL 合成プロジェクトに格納
- ・ 合成パラメータは XST スクリプト ファイルで提供

HDL 合成プロジェクトとは、ASCII テキスト ファイルで、デザインに使用されるさまざまな HDL ソース ファイルがリストされます。拡張子は通常 `.prj` です。次のように 1 つの行で 1 つの HDL ソース ファイルが指定されます。

構文は次のとおりです。

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

この場合、それぞれ次を表します。

- ・ `hdl_language` は、HDL ソース ファイルが VHDL か Verilog かを示します。このフィールドで、VHDL と Verilog の混言プロジェクトの作成を指定できます。
- ・ `compilation_library` は、HDL をコンパイルす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` には HDL ソース ファイルが指定され、パスに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。

HDL プロジェクト ファイルのコード例

次の HDL プロジェクト ファイルの例では、相対パスを使用しています。

```
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
```

XST を ISE® Design Suite から実行すると、プロジェクト ディレクトリに拡張子 `.prj` の付いた HDL プロジェクト ファイルが自動的に作成されます。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プロジェクト ファイルに追加されます。詳細は、ISE Design Suite ヘルプを参照してください。

コマンドラインから XST を起動する場合は、HDL 合成プロジェクト ファイルを手動で作成する必要があります。HDL 合成プロジェクトが XST で読み込まれるようにするには、`run` コマンドラインで入力ファイル名 (`-ifn`) オプションを使用して HDL 合成プロジェクト ファイルのディレクトリを示す必要があります。

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. [File] → [New Project] でプロジェクトを新規に作成します。
2. [Project] → [Add Copy of Source] で HDL ソース ファイルをインポートします。
3. [Design] → [Hierarchy] で最上位レベルのブロックを選択します。
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] をクリックします。
 - c. [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] をクリックするか、[Synthesize - XST] をダブルクリックします。

詳細は、ISE Design Suite ヘルプを参照してください。

コマンドライン モードからの XST の実行

このセクションでは、コマンドラインで XST を起動する方法について説明します。

- ・ [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 のスクリプト ファイル](#)
- ・ [XST のコマンド](#)
- ・ [XST スクリプト ファイルの可読性の改善](#)

XST をスタンドアロン ツールとして実行

XST は、ターミナルまたはコンソール ウィンドウからスタンドアロン 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。

XST をコマンド ライン モードで実行する前に、次の環境変数が正しいザイリンクス ソフトウェアのインストール ディレクトリを指定するように設定します。次は、64 ビット Linux の例です。

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

XST コマンド ライン 構文は、次のようになります。

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

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。

- Linux の場合 **xst** を実行
- Windows の場合、**xst.exe** を実行

XST コマンド ライン オプションには、次のがあります。

- 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。
 - この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。
 - 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。XST をコマンド ライン モードで実行する場合は、[第 15 章「合成レポート」](#)の「**Silent モード**」を参照してください。

XST をインタラクティブに実行

XST を **-ifn** オプションなしで実行した場合、XST コマンド プロンプトに命令を入力できます。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したり、貼り付けたりするよりも、必要なコマンドおよびオプションを含む XST スクリプト ファイルを作成することをお勧めします。XST をスクリプト デザイン インプリメンテーション フローの一部として実行する場合、前もって XST スクリプト ファイルを手動で用意しておくか、自動的に生成させる必要があります。

XST のスクリプト ファイル

XST スクリプト ファイルは、XST コマンドを含む ASCII 形式のテキスト ファイルです。各コマンドには、さまざまなオプションがあります。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 は拡張子 **.xst** で XST スクリプト ファイルを作成します。

XST スクリプト ファイルは、**-ifn** オプションで XST に渡されます。

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

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](#)
- ・ [script コマンド](#)
- ・ [help コマンド](#)

コマンドの中にはオプションを使用して制御できるものがあります。オプションを間違って使用すると、次のようなエラーメッセージが表示されます。

```
ERROR:Xst:1361 - Syntax error in command run for option "-ofmt" :
parameter "EDN" is not allowed.
```

run コマンド

run コマンドは、主要な合成コマンドで、HDL ソース ファイルの解析から最終ネットリストの生成までの合成プロセスすべてを実行します。

run コマンドは、HDL 解析およびエラーボレーションのみを実行する場合にも使用できます。この場合、ソースコードが言語に準拠しているかどうかを確認し、HDL ファイルを前もってコンパイルしたりします。run コマンドは、1 つのスクリプトファイルに対して 1 度だけ使用できます。

構文は次のとおりです。

```
run option_1 value option_2 value ...
```

HDL 記述 (たとえば、デザインの最上位レベル) 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/小文字を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。

次の表には、**run** コマンドの基本的なオプションを示します。ほとんどのオプションは必須です。

run コマンドの基本オプション

オプション	タイプ	コマンドライン	オプション値
入力ファイル名	必須	-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	必須	-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 は削除できます。
ターゲット デバイス	必須	-p	xc6vlx240t-ff1759-1 などの特定のデバイス、または virtex6 などの一般的なデバイス ファミリ名称
最上位モジュール名	必須	-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

オプション	タイプ	コマンドライン	オプション値
VHDL 最上位レベルのアーキテクチャ	オプション	-ent	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。

その他のコマンドライン 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 (タイミング制約以外)」
- ・ 第 13 章「タイミング制約」
- ・ 第 14 章「サポートされるサードパーティ制約」

set コマンド

set コマンドは、**run** 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]
```

set コマンドでは、次の表に示すオプションが使用できます。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
-xsthdpdir	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

script コマンド

XST をインタラクティブ モードで実行する場合、**script** コマンドで XST スクリプト ファイルを読み込んだり、実行したりできます。

構文は次のとおりです。

```
script script_file_name
```

script コマンドでは、XST スクリプト ファイルへのパスを絶対パスでも相対パスでも指定できます。

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ー
- ・ 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

サポートされているファミリを表示するには、引数を指定せずに **help** と入力します。

```
help
```

XST では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
```

特定デバイスの全コマンド

特定のデバイスのコマンドすべてをリストするには、次のように入力します。

```
help -arch family_name
```

この場合、それぞれ次を表します。

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。

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に関する情報を表示するには、次のように入力します。

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

この場合、それぞれ次を表します。

- ・ *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**

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
```

XST スクリプト ファイルの可読性の改善

合成の実行に多数のオプションを使用する場合などは、次のようにファイルを読みやすくします。

- ・ 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。

- **-ifn**
- **-ifmt**
- **-ofn**

- ・ 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)
- 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。

各ディレクトリ名はスペースで区切り、すべてのディレクトリ名を中かっこ {...} で囲む必要があります。次に例を示します。

-vlgincdir {c:\vlg1 c:\vlg2}

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。

- 整数

- ・ 次のようにシャープ文字 (#) を使用すると、そのオプションをコメントアウトしたり、コメントを加えたりすることもできます。。

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 の出力ファイル

このセクションでは、XST から出力される次のファイルについて説明します。

- ・ [XST の通常出力ファイル](#)
- ・ [XST の一時出力ファイル](#)
- ・ [コマンド ライン モードでのスペースを含む名前](#)

XST の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 モードでは、現在のディレクトリまたは **run -ofn** で指定したディレクトリに NGC ファイルが作成されます。
- ・ RTL Viewer 用の Register Transfer Level (RTL) ネットリスト (.ngr)
- ・ 合成ログ ファイル (.srp)
- ・ 一時ファイル

XST の一時出力ファイル

コマンド ライン モードでは、XST の **temp** ディレクトリに一時的なファイルが生成されます。デフォルトでは、XST の **temp** ディレクトリは次になります。

- ・ 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

XST のプロンプトで **set -tmpdir <directory>** を実行するか、XST のスクリプト ファイルで XST の **temp** ディレクトリを変更します。

HDL コンパイル ファイルは、この **temp** ディレクトリに生成されます。デフォルトの **temp** ディレクトリは、現在のディレクトリの下にある **xst** ディレクトリの下です。

この **temp** 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。temp 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 **temp** ディレクトリが自動的にクリーンアップされないで、XST の **temp** 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。

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**、**-vlgindir**) のコマンド ライン 構文も変更されています。これらのオプションで複数のディレクトリを指定する場合は、{...} でディレクトリを囲んでください。

```
-vlgindir {"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。この命名規則は、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、既存のスクリプトは新しい構文に変更することをお勧めします。

VHDL 言語のサポート

メモ: 『XST ユーザー ガイド (Virtex-6 および Spartan-6 用)』は、Virtex®-6 および Spartan®-6 デバイス専用のマニュアルです。その他のデバイスを使用して XST を実行する場合は、『XST ユーザー ガイド』を参照してください。

この章では、VHSIC Hardware Description Language (VHDL) の XST サポートについて説明します。

- ・ [VHDL の利点](#)
- ・ [VHDL の IEEE サポート](#)
- ・ [VHDL のデータ型](#)
- ・ [VHDL のオブジェクト](#)
- ・ [VHDL の演算子](#)
- ・ [VHDL のエンティティとアーキテクチャの記述](#)
- ・ [VHDL の組み合わせ回路](#)
- ・ [VHDL の順序ロジック](#)
- ・ [VHDL の関数とプロシージャ](#)
- ・ [VHDL のアサート文](#)
- ・ [VHDL ライブラリおよびパッケージ](#)
- ・ [VHDL のファイル タイプ サポート](#)
- ・ [VHDL 構文](#)
- ・ [VHDL の予約語](#)

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」](#)
- ・ [第 9 章「デザイン制約」の「VHDL の属性」](#)

VHDL の利点

VHDL は、幅広い言語構文を持ち、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。

- ・ システムをサブシステムに分割する方法や、分割されたサブシステムを相互接続する方法など、システムの構造を記述できます。
- ・ 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簡単に作成できる方法を提供します。この機能により、デザインを効率的に設計でき、タイムトゥ マーケットを短縮できます。

VHDL の IEEE サポート

XST には、VHDL IEEE 1076-1993 に完全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構文
- ・ コードをかなり単純化できる構文
- ・ 合成中に問題が発生しない構文
- ・ 結果に悪影響を及ぼさない構文

たとえば、LRM ではフォーマル ポートが **buffer** で有効なポートが **out** (またはその逆) の場合、ポート マップを使用してインスタンス化ができないことになっていますが、XST ではこのようなインスタンス化がサポートされます。

VHDL のデータ型

このセクションでは、次について説明します。

- ・ [VHDL でサポートされるデータ型](#)
- ・ [VHDL でサポートされないデータ型](#)

次に説明する型には、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 でサポートされるデータ型

このセクションでは、次について説明します。

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

定義済み列挙型

次の定義済み VHDL 列挙型は、次のハードウェア記述でサポートされます。

- 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。
次の表は、使用可能な値と XST でどのように変換されるかをリストしています。

std_logic に使用可能な値

値	意味	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

ユーザー定義の列挙型

列挙型を独自に作成して、次の例のように有限ステート マシン (FSM) のステートを記述することができます。

ユーザー定義の列挙型の VHDL コード例

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

ビット ベクタ型

次のベクタ型は、次のハードウェア記述でサポートされます。

- ・ 標準パッケージで定義される **bit_vector** 型はビット エLEMENTのベクタをモデリングします。
- ・ IEEE **std_logic_1164** パッケージで定義される **std_logic_vector** 型は、**std_logic** ELEMENTのベクタをモデリングします。

次のオーバーロード型も使用できます。

- ・ IEEE **std_logic_1164** パッケージで定義される **std_ulogic_vector** 型
- ・ IEEE **std_logic_1164** パッケージで定義される **unsigned** 型
- ・ IEEE **std_logic_1164** パッケージで定義される **signed** 型

整数型

integer 型は、定義済みの VHDL 型です。デフォルトでは、**integer** は XST 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、**integer** (整数) 型をオーバーロードすることで、この定義済みの **natural** (自然数) および **positive** (正) 型の利点を利用することもできます。

多次元配列型

XST では、何次元でも、次元に制限なく多次元配列型がサポートされますが、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たり、コンポーネント インスタンスエーションで使用されたりします。

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;
```

配列はマトリックスとして宣言することも出来ます。

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

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。

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"));
```

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)
```

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;
```

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

```
MATRIX_A <= CNST_A
```

- ・ 1 行の配列のインデックス

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

- ・ 最大の次元数のインデックス

```
MATRIX_A (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

XST ではレコード型がサポートされます。レコード型は次のように記述できます。

```
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 でサポートされないデータ型

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 のオブジェクト

このセクションでは、次について説明します。

- ・ [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、プロセスまたはサブプログラムで宣言してから、そのプロセスまたはサブプログラム文内で使用します。

VHDL 変数は代入演算子「:=」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 定数は宣言部で宣言でき、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

サポートされる VHDL 演算子については、この章の「[サポートされる VHDL 演算子](#)」セクションに記述しています。このセクションでは、各シフト演算子の使用例を示します。

- ・ SLL (Shift Left Logic) 演算子

```
sig1 <= A(4 downto 0) sll 2
```

これは、次と同じです。

```
sig1 <= A(2 downto 0) & "00";
```

- ・ SRL (Shift Right Logic) 演算子

```
sig1 <= A(4 downto 0) srl 2
```

これは、次と同じです。

```
sig1 <= "00" & A(4 downto 2);
```

- ・ SLA (Shift Left Arithmetic) 演算子

```
sig1 <= A(4 downto 0) sla 2
```

これは、次と同じです。

```
sig1 <= A(2 downto 0) & A(0) & A(0);
```

- ・ SRA (Shift Right Arithmetic) 演算子

```
sig1 <= A(4 downto 0) sra 2
```

これは、次と同じです。

```
sig1 <= <= A(4) & A(4) & A(4 downto 2);
```

- ・ ROL (Rotate Left) 演算子

```
sig1 <= A(4 downto 0) rol 2
```

これは、次と同じです。

```
sig1 <= A(2 downto 0) & A(4 downto 3);
```

- ・ ROR (Rotate Right) 演算子

```
A(4 downto 0) ror 2
```

これは、次と同じです。

```
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

このセクションでは、次について説明します。

- ・ [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 のジェネリックと属性の競合](#)

VHDL の回路記述

VHDL の回路記述 (デザイン ユニット) は、次の 2 つの部分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、外側から見た回路の「外観」が記述されます。
- ・ アーキテクチャ
 - 回路の内部表示を提供
 - 回路のビヘイビアや構造が記述されます。

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**

ポートには通常制約が付きますが、エンティティ宣言で制約を付けないままにしておくこともできます。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。

ただし、ジェネリックを使用して制約を付けたポートを定義し、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使用することをお勧めします。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ジェネリックを宣言できます。詳細については、「[VHDL ジェネリック](#)」を参照してください。

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)

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 では信号が内部および出力ポートとして使用される (内部ドライバが 1 つしかない) 場合にバッファ ポート モードを使用できますが、バッファ ポートは合成中にエラーになる可能性があり、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くなります。

```
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 ポート モードを使用しないコード例 (推奨)

上記の「バッファ モードを使用したコード例 (推奨されない)」では、内部および出力ポートとして使用される信号 C がバッファ モードでモデリングされており、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す。この例では、次のコード例のようにダミー信号を挿入して、C ポートを出力ポートとして宣言することで、バッファ モデルを使用しないようにしています。

```
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 ARCH1 of EXAMPLE is
    signal T : std_logic;
begin
    ...
end ARCH1;
```

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 インスタンス化文のメイン エレメントは次のとおりです。

- ・ **label**
インスタンスを識別します。
- ・ **association list**
 - 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。
- ・ **optional association list**
 - 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 インスタンスレーションの 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。

```
entity NAND2 is
    port (
        A,B : in BIT;
        Y : out BIT );
end NAND2;

architecture ARCH1 of NAND2 is
begin
    Y <= A nand B;
end ARCH1;

entity HALFADDER is
    port (
        X,Y : in BIT;
        C,S : out BIT );
end HALFADDER;

architecture ARCH1 of HALFADDER is
    component NAND2
        port (
            A,B : in BIT;
            Y : out BIT );
    end component
    for all : NAND2 use entity work.NAND2(ARCH1);

    signal S1, S2, S3 : BIT;
begin
    NANDA : NAND2 port map (X,Y,S3);
    NANDB : NAND2 port map (X,S3,S1);
    NANDC : NAND2 port map (S3,Y,S2);
    NANDD : NAND2 port map (S1,S2,S);
    C <= S3;
end ARCH1;
```

VHDL の再帰的なコンポーネント インスタンスレーション文

XST では、再帰的なコンポーネント インスタンスレ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レ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) で制限されます。回数を変更するには、次のコード例のように **-recursion_iteration_limit** オプションを使用します。

再帰的なコンポーネント インスタンス化の VHDL コード例

```

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文

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 (エンティティ/アーキテクチャのペア) とリンクできます。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。次の構文を使用してください。

```

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

```

たとえば、次の文は、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、それが work ライブラリでコンパイルされることを表します。

```

For all : NAND2 use entity work.NAND2(ARCHI);

```

コンポーネント インスタンス文にコンフィギュレーション節がない場合、XST で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、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。

VHDL のジェネリック

VHDL のジェネリックは、Verilog のパラメータと同等のものです。

VHDL のジェネリックを使用すると、拡張可能なデザインを記述するのに役立ちます。ジェネリックは、バス サイズやデザイン ユニットに含まれる一部の反復エレメントの量などの機能をパラメータ化するために使用します。

また、ジェネリックはコンパクトな VHDL コードを記述するためにも使用できます。たとえば、同じ機能をバス サイズを変えて何度もインスタンスエートする必要のある場合、次の「ジェネリック パラメータの VHDL コード例」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。

ジェネリック パラメータは、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 パラメータを使用した VHDL コード例

```
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 (16 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 (16 downto 0);
begin
    U1 : addern generic map (n=>13) port map (X,Y,C1);
    C2 <= C1 & A;
    C3 <= Z & B;
    U2 : addern generic map (n=>17) port map (C2,C3,S);
end bhv;
```

VHDL のジェネリックと属性の競合

VHDL のジェネリックおよび属性は、HDL ソースコードのインスタンスおよびコンポーネントの両方に適用でき、属性は制約ファイルでも指定できますが、競合が発生することもあります。

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。

1. 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。
2. 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、ジェネリックが指定されている箇所に関係なくその属性が考慮されます。サイリンクスでは、同じ制約を定義するのに両方の方法を使用することはお勧めしていません。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。
3. 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優先されます。
4. 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。

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

VHDL の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。これは、アーキテクチャ文の本体で指定できます。

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**)

次の規則が適用されます。

- ・ 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に記述されている順序とは関係ありません。
- ・ 文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

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。

```
--
-- 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 のジェネレート文

このセクションでは、次について説明します。

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

VHDL の for-generate 文

for-generate 文では、繰り返しの構造が記述できます。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。

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。

for-generate 文の VHDL コード例

```
--
-- 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 文は通常ジェネリック値などのテスト結果に基づいて、HDL ソースコードのデバイス特有の部分をアクティベートするために使用します。たとえば、ジェネリックがどのザイリンクス FPGA デバイス ファミリをターゲットにしているか示すために使用されることがあります。この場合、**if-generate** 文により、特定のデバイス ファミリに対するこのジェネリックの値がテストされ、このデバイス ファミリ用に記述された HDL ソースコードのセクションがアクティベートされます。

if-generate 文は、スタティック条件でサポートされます。

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。

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。

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

```
--
-- 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文

このセクションでは、次について説明します。

- ・ [VHDL の組み合わせプロセスの概要](#)
- ・ [VHDL の変数および信号の代入文](#)
- ・ [VHDL の if-else 文](#)
- ・ [VHDL の case 文](#)
- ・ [VHDL の for-loop 文](#)

VHDL の組み合わせプロセスの概要

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。

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、HDL コードを検証して、明示的に代入されていない信号を探します。

組み合わせプロセス文には、**process** 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、センシティビ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

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。

- ・ 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。

この場合、合成結果が最初のデザイン仕様と異なることがあります。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てから、合成を再実行してください。

プロセスには、ローカル変数を含めることができます。

VHDL の変数および信号の代入文

このセクションでは、VHDL の変数および信号の代入文について説明し、次のコード例を提供します。

- ・ 変数および信号代入文の VHDL コード例 1
- ・ 変数および信号代入文の VHDL コード例 2

変数および信号代入文の VHDL コード例 1

次のコード例は、プロセス文内で信号を代入する方法を示しています。

```
entity EXAMPLE is
    port (
        A, B : in BIT;
        S : out BIT );
end EXAMPLE;

architecture ARCH1 of EXAMPLE is
begin
    process (A, B)
    begin
        S <= '0' ;
        if ((A and B) = '1') then
            S <= '1' ;
        end if;
    end process;
end ARCH1;
```

変数および信号代入文の VHDL コード例 2

プロセスには、ローカル変数も含めることができます。変数はプロセス内で宣言して使用でき、通常プロセスの外側からは見えません。

```
entity ADDSUB is
    port (
        A,B : in BIT_VECTOR (3 downto 0);
        ADD_SUB : in BIT;
        S : out BIT_VECTOR (3 downto 0) );
end ADDSUB;

architecture ARCH1 of ADDSUB is
begin
    process (A, B, ADD_SUB)
        variable AUX : BIT_VECTOR (3 downto 0);
    begin
        if ADD_SUB = '1' then
            AUX := A + B ;
        else
            AUX := A - B ;
        end if;
        S <= AUX;
    end process;
end ARCH1;
```

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。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 **if** または **else** 分岐文内で実行できます。**If-else** 文はネストさせることができます。

if-else 文の 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

case 文は論理式を比較し、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

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

この場合、それぞれ次を表します。

◆ *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

```
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 mux4;

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

VHDL の順序ロジック

このセクションでは、次について説明します。

- ・ [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 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、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。この場合、内部ステートまたはメモリ (フリップフロップまたはラッチ) が生成されます。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。

- ・ クロック信号および順次エレメント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) を含むセンシティビティリスト
- ・ クロック イベントを記述した **if** 文
- ・ 非同期制御ロジック (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

構文は次のとおりです。

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

クロック イベント文は、次のように立ち上がりエッジ クロック用に記述できます。

```
If clk'event and clk = '1' then
```

クロック イベント文は、次のように立ち上がりエッジ クロック用に記述できます。

```
If clk'event and clk = '0' then
```

VHDL'93 IEEE 標準の **rising_edge** および **falling_edge** 関数を使用した方がコードはわかりやすくなります。上記の文はそれぞれ次のようになります。

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

XST で信号がセンシティビティリストから削除されたことが検出されなかった場合は、警告メッセージが表示されます。信号がない場合は、リストに自動的に追加されます。ない信号は HDL ソース コードに追加することをお勧めします。ソース コードに追加しないと、シミュレーションからの合成済みのソリューションを有効にしにくくなります。

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、レジスタ、カウンタなどのマクロ推論について説明した第 7 章「HDL コーディング手法」を参照してください。

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** 文を使用した順次ロジックの記述がサポートされます。この場合、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。

- ・ 同じプロセスにはセンシティビティリストと **wait** 文の両方を一緒に使用できません。
- ・ プロセス文に含めることのできる **wait** 文は 1 つだけです。
- ・ **wait** 文は、プロセス文の冒頭に記述します。
- ・ **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、別々にも記述できます。

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

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。センシティビ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、センシティビ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、センシティビ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

VHDL では、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。

- ・ 定数値を指定する必要があります。
- ・ 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) 値を指定しており、順次エレメントは回路に電源が入ってグローバル リセットが適用されると、この値に初期化されます。

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

レジスタを初期化する VHDL コード例 2

順次エレメントは、セット/リセット値およびローカル制御ロジックを記述すると動作に合わせて初期化できます。これには、次の例のようにレジスタのリセットラインの値に対してレジスタの値を指定します。

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

動作に合わせたセット/リセットの詳細、非同期と同期のセット/リセットの比較については、[第 7 章「HDL コーディング手法」](#)の「[フリップフロップおよびレジスタ](#)」を参照してください。

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。

```
entity top is
    Port (
        clk, rst : in std_logic;
        a_in : in std_logic;
        dout : out std_logic);
end top;

architecture behavioral of top is
    signal arb_onebit : std_logic := '1'; -- power-up to vcc
begin
    process (clk, rst)
    begin
        if rst='1' then -- local asynchronous reset
            arb_onebit <= '0';
        elsif (clk'event and clk='1') then
            arb_onebit <= a_in;
        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

ザイリンクス FPGA デバイスに含まれるすべてのメモリエ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。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。この場合、XST は IEEE 規格 (std_logic のデフォルト値は U) に従いません。このような初期化プロセスは、レジスタおよび RAM に対して行われます。

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、次の表の XST 列のようなデフォルト値が使用されます。

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)

未接続の出力ポートは、デフォルトで上記の表の 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 VHDL 規格では、入力ポートを未接続の状態にすることはできません。このため、入力ポートが未接続の場合はエラー 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 の関数とプロシージャ

VHDL の関数 (ファンクション) または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、エンティティの宣言部、アーキテクチャ、またはパッケージで宣言できます。関数またはプロシージャには、宣言部分と本体部分が含まれます。

宣言部分では次が指定されます。

- ・ 入力パラメータ
- ・ 出力と入出力パラメータ (プロシージャのみ)
- ・ 出力と入出力パラメータ (プロシージャのみ)

これらのパラメータ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。レンジューション関数は、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 1

次に、関数をパッケージで宣言するコード例を示します。ここで宣言されている ADD 関数は、1 ビット加算器です。この関数はアーキテクチャ内のパラメータで 4 回呼び出され、4 ビット加算器を作成します。

```
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関数の VHDL コード例 2

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。

```
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 ARCHI 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 ARCHI;
```

プロセスで宣言された関数の VHDL コード例

```
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 ARCHI 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 ARCHI;
```

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

XST では、アサート文がサポートされています。アサート文を使用すると、ジェネリック、定数、generate 条件の不正な値や、呼び出された関数のパラメータの不正な値などの問題を検出して、デザインをデバッグできます。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、警告とその理由が示されるか、エラーとその理由が示されて処理が中断されます。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。

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。assert 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がチェックされます。

SRL のサイズは 16 ビットであり、シフトレジスタの最後のステージはスライスのフリップフロップを使用してインプリメントされるので、シフトレジスタの最大サイズは 17 ビットです。**SINGLE_SRL** ブロックは **TOP** というエンティティで 2 回インスタンス化されます。1 回目の **SRL_WIDTH** は 13、2 回目の **SRL_WIDTH** は 18 です。

デザインルール チェック用 assert 文の VHDL コード例

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。

```
--
-- 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;
```

この場合、XST で次のようなエラー メッセージが表示されます。

```
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

このセクションでは、次の項目について説明します。

- ・ [VHDL ライブラリ](#)
- ・ [VHDL の定義済みパッケージ](#)
- ・ [独自の VHDL パッケージの定義](#)
- ・ [VHDL パッケージのアクセス](#)

VHDL ライブラリ

ライブラリは、デザイン ユニット (エンティティ/アーキテクチャとパッケージ) をコンパイルするディレクトリです。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。

第 2 章「XST プロジェクトの作成および合成」の「HDL 合成プロジェクトの作成」では、HDL 合成プロジェクト ファイルの構文を記述し、HDL ソース ファイルの内容がコンパイルされるライブラリの指定方法について説明しています。

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。

構文は次のとおりです。

```
library library_name ;
```

デフォルトのライブラリは work ライブラリで、この場合 library 節は必要ありません。デフォルトのライブラリ名を変更するには、**run** コマンドで **-work_lib** オプションを使用します。

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、定義済み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、**std** および **ieee standard** ライブラリで定義されているパッケージもサポートされます。これらのライブラリはコンパイル済みで、VHDL コードに直接含めることができるので、ユーザーがコンパイルする必要はありません。定義済みパッケージは、次のとおりです。

- ・ [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 IEEE 実数型および IEEE math_real パッケージ](#)

VHDL の定義済み標準パッケージ

標準 (standard) パッケージには、次の基本的な VHDL 型が含まれます。

- ・ **bit**
- ・ **bit_vector**
- ・ **integer**
- ・ **natural**
- ・ **real**
- ・ **boolean**

標準パッケージはデフォルトで含まれています。

VHDL の定義済み IEEE パッケージ

XST では、さらに多くの共通のデータ型、関数、プロシージャを定義する次の IEEE パッケージがサポートされます。

- ・ **numeric_bit**
bit、オーバーロードされた数値演算子、変換関数、これらの型の拡張関数に基づいてベクタ型 (unsigned および signed) を定義します。
- ・ **std_logic_1164**
std_logic、**std_ulogic**、**std_logic_vector**、**std_ulogic_vector** 型とこれらの型に基づいた変換関数を定義します。
- ・ **std_logic_arith** (Synopsys)
std_logic に基づいて **unsigned** および **signed** ベクタ型を定義します。このパッケージは、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も定義します。
- ・ **numeric_std**
std_logic に基づいて unsigned および signed ベクタ型を定義します。このパッケージは、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も定義します。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を定義します。
- ・ **std_logic_signed** (Synopsys)
std_logic および **std_logic** の符号付き数値演算子を定義します。
- ・ **std_logic_misc** (Synopsys)
std_logic_1164 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、関数を定義します。

IEEE パッケージは、ieee ライブラリであらかじめコンパイルされています。

VHDL の定義済み IEEE 実数型および IEEE math_real パッケージ

標準パッケージの real 型は、IEEE **math_real** パッケージの関数およびプロシージャと同様、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。合成可能な機能を記述するのには、使用できません。

VHDL の実数定数

定数	値	定数	値
math_e	e	math_log_of_2	$\ln 2$
math_1_over_e	$1/e$	math_log_of_10	$\ln 10$
math_pi	π	math_log2_of_e	$\log_2 e$
math_2_pi	2π	math_log10_of_e	$\log_{10} e$
math_1_over_pi	$1/\pi$	math_sqrt_2	$\sqrt{2}$
math_pi_over_2	$\pi/2$	math_1_oversqrt_2	$1/\sqrt{2}$
math_pi_over_3	$\pi/3$	math_sqrt_pi	$\sqrt{\pi}$
math_pi_over_4	$\pi/4$	math_deg_to_rad	$2\pi/360$
math_3_pi_over_2	$3\pi/2$	math_rad_to_deg	$360/2\pi$

VHDL の実数関数

ceil(x)	realmax(x,y)	exp(x)	cos(x)	cosh(x)
floor(x)	realmin(x,y)	log(x)	tan(x)	tanh(x)
round(x)	sqrt(x)	log2(x)	arcsin(x)	arcsinh(x)
trunc(x)	cbrt(x)	log10(x)	arctan(x)	arccosh(x)
sign(x)	"**"(n,y)	log(x,y)	arctan(y,x)	arctanh(x)
"mod"(x,y)	"**"(x,y)	sin(x)	sinh(x)	

独自の VHDL パッケージの定義

次を定義する独自のパッケージを作成できます。

- ・ 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

独自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。

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：上記にリストされるすべてを宣言
- ・ パッケージ本体：パッケージ宣言で宣言された関数およびプロシージャの記述

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

パッケージの定義にアクセスするには、次を実行する必要があります。

- ・ 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。
- ・ 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。

次の構文を使用してください。

```
library library_name ;

use library_name .package_name .all;
```

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

このセクションでは、次について説明します。

- ・ [XST での VHDL ファイルの読み出し/書き込み機能](#)
- ・ [外部ファイルからのメモリ内容の読み込み](#)
- ・ [デバッグ用ファイルへの書き込み](#)
- ・ [書き込み関数を使用したデバッグの規則](#)

XST での 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。

たとえば、ファイル読み込み機能を使用すると、外部ファイルから RAM を初期化できます。

詳細は、第 7 章「HDL コーディング手法」の「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機能は、次の目的に使用できます。

- ・ 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

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**

std_logic_textio パッケージ :

- ・ ieee ライブラリから使用できます。
- ・ 次の表に示すように、**read** および **write** プロシージャをオーバーロードする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。

XST 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

関数	パッケージ
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
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

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内容の読み込み

詳細は、第 7 章「HDL コーディング手法」の「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。

デバッグ用ファイルへの書き込みコード例

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。

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、よくデバッグ目的に使用されます。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 filewrite_explicitopen;

architecture behavioral of filewrite_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の規則

std_logic read の操作で利用できる文字は **0** および **1** 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。**0** と **1** 以外の文字を含むファイルは XST で拒否されます。ただし、ファイル内にスペース文字が検出されると、その文字は無視されます。

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

このセクションでは、次について説明します。

- ・ [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
 - 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** タイプはサポートなし
- ・ 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)

- ・ 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

このセクションでは、次について説明します。

- ・ [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	サポートの有無
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	サポートの有無
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

VHDL では、次の表でサポートなしと記述されている文以外の文がすべてサポートされます。

VHDL の wait 文

wait 文	サポートの有無
Wait on sensitivity_list until Boolean_expression 詳細は、「 VHDL の組み合わせ回路 」を参照してください。	センシティビ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。 メモ : XST では、ラッチの記述に wait 文はサポートされていません。
Wait for time_expression... 詳細は、「 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	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	サポートの有無
for... loop... end loop	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 文でのみサポート
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	サポートの有無
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	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 のサポート

メモ: 『XST ユーザー ガイド (Virtex-6 および Spartan-6 用)』は、Virtex®-6 および Spartan®-6 デバイス専用のマニュアルです。その他のデバイスを使用して XST を実行する場合は、『XST ユーザー ガイド』を参照してください。

この章では、次のセクションに分けて XST の Verilog サポートについて説明します。

- ・ Verilog のサポート
- ・ 変数による部分的ビット選択
- ・ Verilog 構造記述
- ・ Verilog パラメータ
- ・ Verilog パラメータと属性の競合
- ・ Verilog の使用制限
- ・ Verilog 2001 の属性とメタ コメント
- ・ Verilog 構文
- ・ Verilog のタスクおよび関数
- ・ Verilog プリミティブ
- ・ Verilog の予約言語
- ・ Verilog-2001 のサポート

Verilog のサポート

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
- ・ マルチプレクサ
- ・ グルー ロジック
- ・ Finite State Machine (FSM)

Verilog のような Hardware Description Language (HDL) 言語を使用すると、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 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 のような言語を使用してハードウェアを設計すると、並列処理やオブジェクト指向プログラムなど、ソフトウェアの概念を利用できます。Verilog の構文は C 言語および Pascal に類似しており、IEEE 1364 が XST で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。記述されたデザインは、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。数値演算ブロック、グルー ロジック、および Finite State Machine (FSM) コンポーネントには、それぞれ異なる合成方法が使用されます。

このマニュアルは、Verilog の基礎知識を持つ技術者を対象としています。Verilog の詳細は、『IEEE Verilog HDL Reference Manual』を参照してください。

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。

- ・ Verilog デザイン制約とオプション
第 9 章「デザイン制約」
- ・ Verilog 属性の構文
第 4 章「Verilog のサポート」の「Verilog 2001 の属性とメタ コメント」
- ・ 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。
第 10 章「一般制約」

Verilog のビヘイビア記述については、第 5 章「Verilog ビヘイビア記述のサポート」を参照してください。

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 つの明示的な値を指定するのではなく、開始位置およびベクタの幅によって定義されます。開始位置は変更できますが、幅は一定の値が維持されます。

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	意味
+ (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 の構造記述では、複数のブロックを組み合わせ、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は、次のとおりです。

- ・ 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 では、コンポーネントはデザイン モジュールで表されます。

- ・ モジュール宣言では、コンポーネント ポートなどを含め、コンポーネントを外側から見た「外観」が記述されます。
- ・ モジュール本体では、コンポーネントのビヘイビアや構造などの「内部」が記述されます。

コンポーネント間の接続は、コンポーネント インスタンスーション文で定義されます。これらの文では、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。コンポーネント インスタンスーション文はそれぞれ識別子で区別されます。

コンポーネント インスタンスーション文では、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るほか、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、信号やポートをどのコンポーネント宣言のローカル ポートと接続するかが指定されます。

Verilog には多数の論理ゲートが組み込まれており、これらをインスタンスエートして論理回路を作成できます。含まれる論理ゲートは、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

```
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
```

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**

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
```

Verilog の構造記述では、ゲートやレジスタ、CLKDLL や BUFG のようなザイリンクスのプリミティブなど、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、回路を記述することも可能です。これらのプリミティブは Verilog には含まれていませんが、XST Verilog ライブラリ (unisim_comp.v) で提供されています。

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

```
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
```

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

Verilog パラメータ

Verilog パラメータの特徴：

- ・ 簡単に再利用および拡張可能なパラメータ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
- ・ バス サイズやデザイン ユニットに含まれる一部の反復エレメントの量などの機能を記述できる
- ・ Verilog パラメータは制約で、パラメータ モジュールの各インスタンス化の場合、デフォルトのパラメータ値は上書き可能
- ・ VHDL のジェネリックと同等

null 文字列パラメータは、サポートされていません。

ジェネリック (-generics) を使用すると、最上位レベルのブロックで定義される Verilog のパラメータ値を再定義できます。ソースコードを変更しなくてもデザイン コンフィギュレーショ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のようなプロセスで便利な機能です。

Verilog パラメータのコード例

```
module lpm_reg (out, in, en, reset, clk);
    parameter SIZE = 1;
    input in, en, reset, clk;
    output out;
    wire [SIZE-1 : 0] in;
    reg [SIZE-1 : 0] out;

    always @(posedge clk or negedge reset)
    begin
        if (!reset)
            out <= 1'b0;
        else if (en)
            out <= in;
        else
            out <= out; //redundant assignment
    end

endmodule

module top (); //portlist left blank intentionally
    ...
    wire [7:0] sys_in, sys_out;
    wire sys_en, sys_reset, sysclk;

    lpm_reg #8 buf_373 (sys_out, sys_in, sys_en, sys_reset, sysclk);
    ...
endmodule
```

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。

Verilog パラメータと generate-for のコード例

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。

次の例は、パラメータと **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、「[Verilog ビヘイビア記述の 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 パラメータと属性の競合

競合の理由には、次のとおりです。

- ・ パラメータおよび属性は 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

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。

1. 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。
2. パラメータ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、パラメータが優先され、XST で警告メッセージが表示されます。
3. 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。

Verilog パラメータと属性の優先順位

	インスタンスのパラメータ	モジュールのパラメータ
インスタンスの属性	パラメータを適用(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	パラメータを適用	パラメータを適用(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

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よりも優先されます。

Verilogの使用制限

このセクションでは、XST の Verilog サポートの制限、ザイリンクスの推奨するサポート機能の制限について説明します。このセクションには、次の項目が含まれます。

- ・ [大文字/小文字の区別](#)
- ・ [ブロッキング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、Verilog の大文字/小文字の区別が完全にサポートされますが、潜在的な名前の競合については注意してください。

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問題なく合成されますが、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、混合言語プロジェクトで問題になったり、制約を XCF ファイルを使用して適用できなくなります。

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の両方が完全にサポートされます。

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](#)
- ・ [連結文での整数処理](#)

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。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、例の後半のように 4 を 3 ビットに指定します。

case 文での整数処理のコード例

```
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

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。結果がビット指定のない整数となる式を使用する場合は、次に示すように一時的な信号 (temp) を指定し、その一時信号を連結文で使用します。

連結文での整数処理のコード例

```
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

XST では、次がサポートされます。

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

Verilog-2001 の属性

XST では Verilog-2001 属性文がサポートされています。属性は、合成ツールなどのソフトウェア ツールに特定の情報を渡すために使用します。Verilog-2001 の属性はよく使用されてきているので、使用をお勧めしています。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。その他の属性宣言はコンパイラでサポートされる場合がありますが、XST では無視されます。

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
 - **module**
 - **instance**
 - **net**
- ・ 次の合成制約を設定する場合
 - [フル ケース \(FULL_CASE\)](#)
 - [パラレル ケース \(PARALLEL_CASE\)](#)

Verilog のメタ コメント

メタ コメントとは、Verilog の解析で認識されるコメントのことです。Verilog メタ コメントは、次のために使用します。

- ・ 次のような個々のオブジェクトに制約を設定する場合
 - **module**
 - **instance**
 - **net**
- ・ 合成で次のような制約を設定します。
 - 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。

メタ コメントは次のスタイルを使用して記述できます。

- ・ 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。

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
- ・ 変換なし (TRANSLATE_OFF) と変換あり (TRANSLATE_ON)

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

- ・ パラレル ケース (PARALLEL_CASE)

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

- ・ 各オブジェクトに対する制約

一般的な構文は次のとおりです。

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

Verilog メタ コメントのコード例

```
// synthesis attribute RLOC of ul23 is R11C1.S0  
// synthesis attribute HUSSET ul MY_SET  
// synthesis attribute fsm_extract of State2 is "yes"  
// synthesis attribute fsm_encoding of State2 is "gray"
```

Verilog 構文

このセクションでは、サポートされる Verilog 構文とサポートされない Verilog 構文について次に分けて説明します。

- ・ 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

メモ : XST では、信号名の冒頭文字にアンダースコアを使用できません (`_DATA_1` など)。

Verilog の定数

XST では次の表に示していない限り、すべての Verilog 定数がサポートされます。

サポートされる Verilog の定数

定数	サポートの有無
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

XST では次の表に示していない限り、すべての Verilog データ型がサポートされます。

XST でサポートされる Verilog のデータ型

データ型	カテゴリ	サポートの有無
ネットタイプ	tri0、tri1、triereg	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

XST では次の表に特に書いていない限り、すべての Verilog 継続代入文がサポートされます。

サポートされる Verilog の継続代入文

継続代入文	サポートの有無
駆動電流	無視
遅延	無視

Verilog の手続き代入文

XST では次の表に示していない限り、すべての Verilog 手続き代入文がサポートされます。

サポートされる Verilog の手続き代入文

手続き代入文	サポートの有無
assign	制限付きでサポートあり。詳細は、「Verilog ビヘイビア記述の 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、「Verilog ビヘイビア記述の 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
名前付きイベント	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

XST では次の表に示していない限り、すべての Verilog デザイン階層がサポートされます。

XST でサポートされる Verilog のデザイン階層

デザイン階層	サポートの有無
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

XST では次の表に特に示していない限り、すべての Verilog コンパイラ指示子がサポートされます。

サポートされる Verilog のコンパイラ指示子

コンパイラ指示子	サポートの有無
'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
'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 のタスクおよび関数

このセクションでは、次について説明します。

- ・ サポートされるシステム タスクと関数
- ・ 変換関数の使用
- ・ ファイル I/O タスクを使用したメモリ内容の読み込み
- ・ ディスプレイ タスク
- ・ \$finish を使用したデザイン ルール チェックの作成

サポートされるシステム タスクと関数

サポートされるシステム タスクと関数

システム タスクと関数	サポートの有無	コメント
<code>\$display</code>	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。
<code>\$fclose</code>	サポートあり	
<code>\$fdisplay</code>	サポートあり	
<code>\$fgets</code>	サポートあり	
<code>\$finish</code>	サポートあり	<code>\$finish</code> 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。
<code>\$fopen</code>	サポートあり	
<code>\$fscanf</code>	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。
<code>\$fwrite</code>	サポートあり	
<code>\$monitor</code>	無視	
<code>\$random</code>	無視	
<code>\$readmemb</code>	サポートあり	
<code>\$readmemh</code>	サポートあり	
<code>\$signed</code>	サポートあり	
<code>\$stop</code>	無視	
<code>\$strobe</code>	無視	
<code>\$time</code>	無視	
<code>\$unsigned</code>	サポートあり	
<code>\$write</code>	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	

サポートされないシステム タスクは、XST の Verilog コンパイラで無視されます。

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)      $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、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。詳細は、[第 7 章「HDL コーディング手法」](#)の「[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。

2 進数の場合は `$readmemb`、16 進数の場合は `$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、これらのシステム タスクでは次のようにインデックス パラメータを使用することをお勧めします。

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

ディスプレイ タスク

ディスプレイ タスクは、情報をコンソールに表示したり、外部ファイルに書き出したるするために使用できます。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。XST では、次のエスケープ シーケンスのサブセットがサポートされます。

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

`$display` の Verilog 構文例

次に、2 進数の定数値を 10 進数で表示する `$display` の構文を示すコード例を示します。

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

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

`$finish` シミュレーション コントロール タスクは主にシミュレーション用で、XST ではビルトインのデザイン ルール チェックを作成するために一部サポートされます。デザイン ルール チェックでは、正しい構文で記述されたデザイン コンフィギュレーションが検出されますが、機能しなかったり、希望通りのインプリメンテーションにならないこともあります。**`$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るので、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 ツールのみで、XST も含めた合成ツールではこれらが無視されます。

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。

\$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
```

ダイナミックにアクティブな状況で **\$finish** システム タスクが使用されると、それを示すメッセージが表示され、タスクは無視されます。

\$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。このようなスタティックに評価される状況の場合、主にパラメータと予測値が比較されます。これは、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。

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
end

always @(posedge clk)
begin
    do <= di;
end

endmodule
```

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。

Verilog プリミティブ

XST では、次のように Verilog プリミティブがサポートされています。

- ・ XST では、一部のゲートレベル プリミティブがサポートされています。サポートさる構文は次のとおりです。
`gate_type instance_name (output, inputs,...);`
 次に、ゲートレベル プリミティブのインスタンス化の例を示します。
`and U1 (out, in1, in2); bufif1 U2 (triout, data, trienable);`
- ・ 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 でサポートされる Verilog のゲート レベルのプリミティブ

プリミティブ	サポートの有無
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

- ・ 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 では、Verilog のユーザー定義のプリミティブはサポートされません。

Verilog の予約言語

次の表は、Verilog の予約語を示しています。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。

Verilog の予約言語

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
pulsetype- _ondetect*	pulsetype- _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	triereg	use*
vectored	wait	wand	weak0
weak1	while	wire	wor
xnor	xor		

Verilog2001 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。Verilog 2001 の詳細については、Stuart Sutherland 著『Verilog-2001: A Guide to the New Features』または『IEEE Standard Verilog Hardware Description Language』(IEEE Standard 1364-2001) を参照してください。

- ・ 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 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
- ・ インライン パラメータの明示
- ・ 固定ローカル パラメータ
- ・ 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

Verilog ビヘイビア記述のサポート

メモ: 『XST ユーザー ガイド (Virtex-6 および Spartan-6 用)』は、Virtex®-6 および Spartan®-6 デバイス専用のマニュアルです。その他のデバイスを使用して XST を実行する場合は、『XST ユーザー ガイド』を参照してください。

この章では、次のセクションに分けて XST の Verilog ビヘイビア記述のサポートについて説明します。

- ・ Verilog ビヘイビア記述の変数宣言
- ・ Verilog ビヘイビア記述の初期値
- ・ Verilog ビヘイビア記述の配列のコード例
- ・ Verilog ビヘイビア記述の多次元配列
- ・ Verilog ビヘイビア記述のデータ型
- ・ Verilog ビヘイビア記述で使用可能な文
- ・ Verilog ビヘイビア記述の論理式
- ・ Verilog ビヘイビア記述のブロック
- ・ Verilog ビヘイビア記述のモジュール
- ・ Verilog ビヘイビア記述の継続代入文
- ・ Verilog ビヘイビア記述の手続き代入文
- ・ Verilog ビヘイビア記述のタスクおよび関数
- ・ Verilog ビヘイビア記述のブロックおよびノンブロッキング手続き代入文
- ・ Verilog ビヘイビア記述の定数
- ・ Verilog ビヘイビア記述のマクロ
- ・ Verilog ビヘイビア記述の include ファイル
- ・ Verilog ビヘイビア記述のコメント
- ・ Verilog ビヘイビア記述の generate 文

Verilog ビヘイビア記述の変数宣言

Verilog の変数は、整数 (integer) または実数 (real) として宣言できます。ただし、これらの宣言はテストコードで使用するためのものです。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;
```

Verilog ビヘイビア記述の初期値

Verilog-2001 では、レジスタを宣言する際に初期値を設定できます。指定される初期値は、次の通りです。

- ・ 定数値を指定する必要があります。
- ・ 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。このように初期値を指定すると、レジスタの INIT 属性として NGC ファイルに記述されます。これらの値は、ローカルリセットとは関係ありません。

Verilog ビヘイビア記述の初期値のコード例 1

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

また、ビヘイビア記述の Verilog コードを使用して、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の値に対してレジスタの値を指定するには、次の例のように記述します。

Verilog ビヘイビア記述の初期値のコード例 2

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

ビヘイビアコードで変数の初期値を設定すると、出力がローカルリセットで制御可能なフリップフロップとしてデザインにインプリメントされ、NGC ファイルに FDP または FDC フリップフロップとして記述されます。

Verilog ビヘイビア記述の配列のコード例

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。

Verilog ビヘイビア記述の配列のコード例 1

次の Verilog 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];
```

Verilog ビヘイビア記述の配列のコード例 2

次の例は、8 ビット幅のELEMENTが 64 個ある配列を示しており、Verilog の構造記述で次のように指定できます。

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

Verilog ビヘイビア記述の多次元配列

XST では、2 次元までの多次元配列型がサポートされます。多次元配列はネットまたはさまざまなデータ型で使用できます。配列を使用して代入および数値演算を記述できますが、一度に選択できる配列の要素は 1 つのみです。システム タスクまたは関数、通常のタスクまたは関数で多次元配列を使用することはできません。

多次元配列の Verilog ビヘイビア記述のコード例 1

次の例は、8 ビット幅の wire エlement を 256 X 16 個含む配列を示しており、Verilog の構造記述でのみ指定できます。

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

多次元配列の Verilog ビヘイビア記述のコード例 2

次の例は、64 ビット幅のレジスタ エlement を 256 X 8 個含む配列を示しており、Verilog のビヘイビア記述でのみ指定できます。

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

Verilog ビヘイビア記述のデータ型

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 では、次の Verilog データ型がサポートされます。

- ・ **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 statel = 3'b001; // 3 bit constant
parameter component = "TMS380C16"; // string
```

Verilog ビヘイビア記述で使用可能な文

次に、Verilog のビヘイビア記述で使用可能な文 (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。

Verilog ビヘイビア記述の論理式

このセクションでは、Verilog ビヘイビア記述の論理式について説明します。

- ・ [Verilog ビヘイビア記述の論理式の概要](#)
- ・ [Verilog ビヘイビア言語でサポートされる演算子](#)
- ・ [Verilog ビヘイビア記述でサポートされる論理式](#)
- ・ [Verilog ビヘイビア記述の論理式の評価結果](#)

Verilog ビヘイビア記述の論理式の概要

論理式では、数値演算、論理演算、関係演算、条件演算で定数および変数が演算されます。論理演算は、複数のビットまたは 1 つのビット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。

Verilog ビヘイビア言語でサポートされる演算子

次の表は、ビヘイビア記述でサポートされる Verilog 演算子を示しています。

Verilog ビヘイビア言語でサポートされる演算子

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

Verilog ビヘイビア記述でサポートされる論理式

次の表は、ビヘイビア記述でサポートされる Verilog 論理式を示しています。

Verilog ビヘイビア記述でサポートされる論理式

論理式	シンボル	サポートの有無
接続	{}	サポートあり
複製	{}	サポートあり
数値演算	+, -, *, **	サポートあり
除算	/	2 つ目のオペランドが 2 のべき乗の場合 または両方のオペランドが定数の場合にサポート
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	サポートの有無
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

Verilog のビヘイビア記述の論理式の評価結果

次の表は、よく使用される演算子を使用した評価論理式を示しています。=== と !== は、特別な比較演算子で、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。

Verilog のビヘイビア記述の論理式の評価結果

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

Verilog ビヘイビア記述のブロック

ブロック文は、複数の文をグループ化します。XST では、順次ブロックのみがサポートされています。ブロック内では、記述された順に文が実行されます。ブロックは begin と end キーワードで示されます。これについては、この章の後の方で例を示します。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。

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

各文は手続き代入文であり、セミコロンで区切られます。

Verilog ビヘイビア記述のモジュール

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。このセクションでは、次のセクションに分けてビヘイビア記述の Verilog モジュールについて説明します。

- ・ Verilog ビヘイビア記述のモジュール宣言
- ・ Verilog ビヘイビア記述のモジュール インスタンス化

Verilog ビヘイビア記述のモジュール宣言

Verilog ビヘイビア記述のモジュールは、次のコード例のように宣言されます。

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

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。

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 : input、output、inout
- ・ 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 2

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

endmodule

Verilog ビヘイビア記述のモジュール インスタンス化

Verilog ビヘイビア記述のモジュールは、別のモジュールに次のようにインスタンス化されます。

Verilog ビヘイビア記述のモジュール インスタンスエーションのコード例

```
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
```

モジュール インスタンスエーション文では、インスタンス名を定義します。この文には、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を指定するポート関連リストが含まれます。リストのエレメントは、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 ビヘイビア記述の継続代入文

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。assign 文を使用する代入と使用しない代入の両方がサポートされています。

- ・ 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;
```

- ・ 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;
```

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。

Verilog ビヘイビア記述の手続き代入文

このセクションでは、Verilog ビヘイビア記述の手続き代入文について説明します。

- ・ [Verilog ビヘイビア記述の手続き代入文の概要](#)
- ・ [組み合わせ always ブロック](#)
- ・ [if - else 文](#)
- ・ [case 文](#)
- ・ [for および repeat ループ文](#)
- ・ [while 文](#)
- ・ [順次 always ブロック](#)
- ・ [assign 文および deassign 文](#)
- ・ [32 ビットを超える場合のビットの拡張](#)

Verilog ビヘイビア記述の手続き代入文の概要

手続き代入文は、次のように使用されます。

- ・ `reg` として宣言された変数に値を代入するために使用されます。
- ・ `always` ブロック、タスク、関数で最初に使用されます。
- ・ 通常はレジスタおよび Finite State Machine (FSM) を記述するために使用されます。

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、次の Verilog のタイミング制御文を使用して効率的に記述できます。

- ・ 遅延 : `#` (シャープ記号)
- ・ イベント制御 : `@` (アット マーク)

遅延時間の制御文は、シミュレーションでのみ考慮され、合成ツールでは無視されます。

合成では `#` によるタイミング制御は無視されるため、ここでは `@` 文を使用した組み合わせロジックの記述を説明します。

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。このセンシティビティリストには、条件 (`if`, `case` など) となり得る信号、および代入文の右側に記述される信号を含むことができます。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

`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

次に、if - else 文を使用してマルチプレクサを記述した例を示します。

```
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

case 文は論理式を比較し、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すると、予測不可能な結果になります。必ず整数のサイズをビット数で指定してください。
- ・ 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 文のコード例

次に、case 文を使用してマルチプレクサを記述した例を示します。

```
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
        case (sel)
            2'b00: outmux = a;
            2'b01: outmux = b;
            2'b10: outmux = c;
            default: outmux = d;
        endcase
    end
endmodule
```

この 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。優先順に評価されるのを防ぐには、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 ループ文

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 for 文または repeat 文も使用できます。

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**この場合、それぞれ次を表します。
 - **var** はループ変数
 - **step** は定数値

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 文

always ブロックでは、while 文を使用して繰り返し処理を実行できます。while 文は、テスト式が偽 (false) になるまで、含まれる文を実行します。テスト式が始めから false の場合は実行されません。

- ・ 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 文の構文は、ラベルが付いているブロック内に次の構文で使用します。

disable <blockname>

```
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
```

順次 always ブロック

always ブロックと次のエッジトリガ イベント (posedge または negedge) を含むセンシティビ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モデリング)

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 à )
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

次の例では、アクティブ High の非同期リセットとアクティブ Low の非同期セットを示しています。

```
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 コード例 4

次の例では、非同期セット/リセットを含まず、同期リセットを含むレジスタを示しています。

```
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

assign 文および deassign 文はサポートされていません。

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、左側が次の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。

上記の規則は、Verilog-2001 標準に従っています。これらは、Verilog-1995 との互換性はありません。

Verilog ビヘイビア記述のタスクおよび関数

このセクションでは、Verilog ビヘイビア記述のタスクおよび関数について説明します。

- ・ [Verilog ビヘイビア記述のタスクおよび関数](#)
- ・ [Verilog ビヘイビア記述のタスクおよび関数のコード例](#)
- ・ [Verilog ビヘイビア記述の再帰タスクおよび関数](#)
- ・ [Verilog ビヘイビア記述の定数関数](#)

Verilog ビヘイビア記述のタスクおよび関数

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、コードの量を削減したり、維持がしやすくなります。

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。ヘッダ部には次のパラメータが含まれます。

- ・ 入力パラメータ (関数の場合のみ)
- ・ 入力/出力/入出力パラメータ (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できます。内容は組み合わせ always ブロック文に類似しています。

Verilog ビヘイビア記述のタスクおよび関数のコード例

このセクションでは、Verilog ビヘイビア記述のタスクおよび関数のコード例について説明します。

- ・ タスクおよび関数のコード例 1
- ・ タスクおよび関数のコード例 2

Verilog ビヘイビア記述のタスクおよび関数のコード例 1

次の例では、1 ビット加算器を宣言する ADD 関数がアーキテクチャ内のパラメータで 4 回呼び出され、4 ビット加算器が作成されています。

```
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

次の例では、同じ動作をタスクを使用して記述した例です。

```
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
```

Verilog ビヘイビア記述の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。

Verilog ビヘイビア記述の反復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
```

Verilog ビヘイビア記述の定数関数

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 Verilog ビヘイビア記述のコード例

```
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
```

Verilog ビヘイビア記述のブロックおよびノンブロッキング手続き代入文

タイミング制御文の # および @ を使用すると、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。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。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 Verilog ビヘイビア記述のコード例 1

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

ブロッキング手続き代入文の Verilog ビヘイビア記述のコード例 2 (代替方法)

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

このタイプの代入文では、文が 1 つずつ順に実行され、プロセスに含まれる別の文が同時に実行されることはありません。これは、主にシミュレーションで使用します。

ノンブロッキング代入文では、文が実行されるときに式が評価されますが、同じプロセスに含まれるほかの文も同時に実行されます。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 Verilog ビヘイビア記述のコード例 1

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

ノンブロッキング手続き代入文の Verilog ビヘイビア記述のコード例 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;
```

Verilog ビヘイビア記述の定数

Verilog の定数は、デフォルトでは 10 進整数と認識されますが、適切な接頭辞を使用して 2 進、8 進、10 進、16 進に指定できます。たとえば、次はすべて同じ値を表します。

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

Verilog ビヘイビア記述のマクロ

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、次のようにデザイン コードの後で参照されます。

```
if (request == 'TESTEQ1)
```

Verilog ビヘイビア記述のマクロのコード例 1

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

Verilog では、マクロが定義されているかどうかを判断する 'ifdef および 'endif も使用できます。これらの構文は、条件付きコンパイルを定義するために使用します。'ifdef コマンド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 ビヘイビア記述のマクロのコード例 2

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

Verilog マクロ (-define) を使用すると、Verilog マクロを定義または再定義でき、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。これは、IP コアの生成やフロー テストのようなプロセスで便利な機能です。

Verilog ビヘイビア記述の 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。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、複数ファイルでデザインに含まれる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。

- ・ ISE Design Suite はデフォルトでプロジェクト ディレクトリを検索するので、ファイルをプロジェクト ディレクトリに追加すると ISE Design Suite で認識されるようになります。
- ・ 相対パスまたは絶対パスを HDL ソース コードの 'include 文に含めることで別のディレクトリを ISE Design Suite に認識させることができます。
- ・ XST が直接 include ファイル ディレクトリをポイントするようにするには、[Verilog インクルード ディレクトリ \(-vlgincdir\)](#) を使用します。
- ・ ISE Design Suite でデザイン階層を構築するのにこのインクルード ファイルが必要とされる場合は、プロジェクト ディレクトリに含めるか、または相対パスまたは絶対パスで参照させる必要があります。ファイルをプロジェクトに追加する必要はありません。

XST デザイン プロジェクト ファイルでは、別の方法で Verilog ファイルの内容をプロジェクトの残りに対して認識させます。ザイリンクスでは、XST 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が、このインクルード方法を使用する場合は、競合に気をつけてください。ここで記述する以外の方法で Verilog ファイルを含めたり、そのときに XST デザイン プロジェクト ファイルをリストしたりすると、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 ファイルをそのような方法で ISE Design Suite プロジェクトに追加した場合も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

XST では、次の形式の Verilog ビヘイビア記述のコメントがサポートされます。

- ・ コメントが 1 行の場合は // で開始します。

```
// This is a one-line comment
```
- ・ 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
*/
```

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。

Verilog ビヘイビア記述の generate 文

このセクションでは、Verilog ビヘイビア記述の generate 文について説明します。

- ・ Verilog ビヘイビア記述の generate 文
- ・ Verilog ビヘイビア記述の generate ループ文
- ・ Verilog ビヘイビア記述の generate 条件文
- ・ generate ケース文

Verilog ビヘイビア記述の generate 文

generate 文を使用すると、パラメータ変更可能なスケーラブルなコードを作成できます。generate 文の内容は条件別にデザインにインスタンスエートできます。generate 文は Verilog のエラボレーション中に実行されます。

generate 文を使用すると、反復的な構文やスケーラブルな構文を作成したり、特定の条件を満たす条件関数を作成することもできます。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
- ・ パラメータの再定義
- ・ タスクまたは関数の定義

generate 文はモジュール内で記述し、generate で開始し、endgenerate で終了します。

XST では、次の 3 つの generate 文がサポートされています。

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

Verilog ビヘイビア記述の 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 Verilog ビヘイビア記述のコード例

```
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
```

Verilog ビヘイビア記述の 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 条件文の Verilog ビヘイビア記述のコード例

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エートしています。

```
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
```

Verilog ビヘイビア記述の 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エートしています。

```
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

メモ: 『XST ユーザー ガイド (Virtex-6 および Spartan-6 用)』は、Virtex®-6 および Spartan®-6 デバイス専用のマニュアルです。その他のデバイスを使用して XST を実行する場合は、『XST ユーザー ガイド』を参照してください。

この章では、XST の混合言語サポートについて説明します。

- ・ [混合言語サポートの概要](#)
- ・ [VHDL/Verilog の境界規則](#)
- ・ [ポートのマッピング](#)
- ・ [ジェネリックのサポート](#)
- ・ [Library Search Order \(LSO\) ファイル](#)

混合言語サポートの概要

XST では、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 の混合は、デザイン ユニット (セル) のインスタンス化のみ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 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、エラボレーション段階で 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。
- ・ 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。
- ・ Verilog モジュールを VHDL にインスタンス化する場合、コンフィギュレーションの設定、直接のインスタンス化、およびコンポーネント コンフィギュレーションはサポートされません。
- ・ プロジェクトを構成する VHDL および Verilog ファイルは、独自の XST HDL プロジェクト ファイルで指定します。XST HDL プロジェクトの指定方法については、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。
- ・ 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。
- ・ 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する xhdp.ini のメカニズムは、VHDL でも Verilog でも使用できます。
- ・ 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

このセクションでは、VHDL および Verilog の境界規則について説明します。

- ・ [VHDL/Verilog の境界規則](#)
- ・ [VHDL への Verilog モジュールのインスタンス化](#)
- ・ [Verilog デザインへの VHDL デザイン ユニットのインスタンス化](#)

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化でき、Verilog のモジュールには VHDL エンティティをインスタンス化できます。

Verilog デザインへの VHDL デザイン ユニットのインスタンスシート

VHDL エンティティをインスタンスシートするには、次の手順に従ってください。

1. インスタンスシートする 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 インスタンスシエーションを実行

Verilog デザインにインスタンスシートできる 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。

XST では、バインド処理はエラボーレーション段階で行われます。バインド処理では、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、Verilog モジュール名 (モジュール インスタンスシエーションで指定されたアーキテクチャ名は無視される) が検索されます。詳細は、「[Library Search Order \(LSO\) ファイル](#)」を参照してください。

見つかった場合は、その名前がバインドされます。Verilog モジュールが見つからない場合は、インスタンスシートされたモジュールの名前は VHDL エンティティとして扱われ、大文字と小文字を区別して検索が行われます。VHDL エンティティは、VHDL デザイン ユニットが拡張識別子付きで保存されていると仮定して、ユーザー指定の順序でライブラリのユーザー指定のリストから検索されます。詳細は、「[Library Search Order \(LSO\) ファイル](#)」を参照してください。見つかった場合は、その名前がバインドされます。XST では、最初に一致した VHDL エンティティを選択してバインドします。

Verilog モジュールから VHDL デザイン ユニットのインスタンスシートする場合、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。
- ・ パラメータは、値が変化しない場合でも、インスタンスシート時にすべて渡す必要があります。
- ・ パラメータを変更する場合は、どのパラメータ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シエーションを使用してください。

使用可能なコード例

次のコード例は使用できます。

```
ff #(.init(2'b01)) u1 (.sel(sel), .din(din), .dout(dout));
```

使用不可なコード例

次のコード例は使用できません。

```
ff u1 (.sel(sel), .din(din), .dout(dout));
defparam u1.init = 2'b01;
```

VHDL への Verilog モジュール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。Verilog モジュール名がすべて小文字でない場合は、次のいずれかの方法で case プロパティを使用し、大文字/小文字を保持するように設定します。
 - ・ ISE® Design Suite
[Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスを表示して、[Synthesis Options] → [Case] → [Maintain] を選択します。
 - ・ コマンドライン
-case を maintain に設定
2. 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 ユニットに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化できる 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。

- ・ エラボ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。
 - VHDL デザイン ユニットが見つからない場合は、コンポーネント名は Verilog モジュールとして扱われ、大文字と小文字を区別して検索が行われます。

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。

ポートのマップ

このセクションでは、ポートのマップについて説明します。

- ・ [Verilog への VHDL のインスタンス化](#)
- ・ [VHDL への Verilog のインスタンス化](#)

Verilog への VHDL のインスタンス化

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 への Verilog のインスタンシエート

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シエートされる場合、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**
- ・ XST では、Verilog の双方向パス オプションはサポートされていません。
- ・ XST では、混合言語の境界に名前のない Verilog ポートを使用することはできません。
- ・ 使用可能なデータ型 :
 - **wire**
 - **reg**

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。

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**

LSO ファイル

このセクションでは、Library Search Order (LSO) ファイルについて説明します。

- ・ [LSO の概要](#)
- ・ [ISE® Design Suite での LSO ファイルの指定](#)
- ・ [コマンド ライン モードでの LSO ファイルの指定](#)
- ・ [LSO の規則](#)

LSO ファイルの概要

ライブラリ検索順ファイル (Library Search Order (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

ISE® Design Suite では、Library Search Order (LSO) ファイルのデフォルト名は `project_name.lso` です。`project_name.lso` ファイルが存在しない場合は、ISE Design Suite により自動的に作成されます。ISE Design Suite で既存の `project_name.lso` ファイルが検出されると、そのファイルがそのまま使用されます。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。ISE Design Suite でデフォルトの LSO ファイルが作成されると、ファイルの最初の行に **DEFAULT_SEARCH_ORDER** キーワードが記述されます。

コマンドライン モードでの LSO ファイルの指定

コマンドラインから XST を使用する場合は、Library Search Order (LSO) (**-lso**) オプションを使用して LSO ファイルを指定します。**-lso** オプションを使用しない場合は、LSO ファイルを使用せずに、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

Library Search Order (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 ファイルに表示される順にライブラリ リストが追加され、LSO ファイルがアップデートされます。

DEFAULT_SEARCH_ORDER キーワードのみの場合

Library Search Order (LSO) ファイルに **DEFAULT_SEARCH_ORDER** キーワードのみのが含まれる場合、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 ファイルが次のようにアップデートされます。
 - **DEFAULT_SEARCH_ORDER** キーワードが削除される
 - 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る

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   rtfl1lib f1.v
vhdl      vhlib2   f3.vhd
```

LSO ファイル `my_proj.lso` の内容は、次のとおりです。

```
DEFAULT_SEARCH_ORDER
```

XST では、次の検索順が使用されます。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。

```
vhlib1
rtfllib
vhlib2
```

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

Library Search Order (LSO) ファイルに DEFAULT_SEARCH_ORDER キーワードとライブラリのリストが含まれる場合、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。

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。

```
vhd1      vhlib1   f1.vhd
verilog   rtfllib  f1.v
vhd1      vhlib2   f3.vhd
```

LSO ファイル my_proj.lso の内容は、次のとおりです。

```
rtfllib
vhlib2
vhlib1
DEFAULT_SEARCH_ORDER
```

XST では、次の検索順が使用されます。

```
vhlib1
rtfllib
vhlib2
```

プロセス後の my_proj.lso の内容に変更はありません。

```
rtfllib
vhlib2
vhlib1
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。

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。

```
vhd1 vhlib1 f1.vhd
verilog rtfllib f1.v
vhd1 vhlib2 f3.vhd
```

LSO ファイル my_proj.lso の内容は、次のとおりです。

```
rtfllib  
vhlib2  
vhlib1
```

XST では、次の検索順が使用されます。

```
rtfllib  
vhlib2  
vhlib1
```

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

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 Library Search Order (LSO) ファイルに含まれており、LSO ファイルに DEFAULT_SEARCH_ORDER キーワードが含まれていない場合、そのライブラリは無視されます。

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
```

LSO ファイル my_proj.lso が次の内容で作成されます。

```
personal_lib  
rtfllib  
vhlib2  
vhlib1
```

XST では、次の検索順が使用されます。

```
rtfllib  
vhlib2  
vhlib1
```

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

メモ: 『XST ユーザー ガイド (Virtex-6 および Spartan-6 用)』は、Virtex®-6 および Spartan®-6 デバイス専用のマニュアルです。その他のデバイスを使用して XST を実行する場合は、『XST ユーザー ガイド』を参照してください。

この章には、次の内容が含まれます。

- ・ HDL コーディング手法の概要
- ・ 記述言語の選択
- ・ マクロ推論フローの概要
- ・ フリップフロップおよびレジスタ
- ・ ラッチ
- ・ トライステート
- ・ カウンタおよびアキュムレータ
- ・ シフトレジスタ
- ・ ダイナミック シフトレジスタ
- ・ マルチプレクサ
- ・ 四則演算
- ・ コンパレータ
- ・ 除算器
- ・ 加算器、減算器、加減算器
- ・ 乗算器
- ・ 乗加算および乗累算
- ・ DSP の推論
- ・ リソース共有
- ・ RAM
- ・ ROM
- ・ 有限ステート マシン (FSM) コンポーネント
- ・ ブラック ボックス

HDL コーディング手法の概要

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。
- ・ Virtex®-6 と Spartan®-6 デバイス アーキテクチャの機能を利用できます。

この章のほとんどのセクションには、次が記述されています。

- ・ 機能の一般的な説明
- ・ HDL ソース コードでその機能を使用するガイドライン
- ・ XST で Virtex-6 と Spartan-6 デバイスにその機能がどのようにインプリメントされるかの情報。詳細は、[第 8 章「FPGA の最適化」](#)を参照してください。
- ・ XST でのその機能の処理方法を制御する制約のリスト
- ・ レポート例
- ・ VHDL および Verilog コード例

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。

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。

記述言語の選択

次の表は、VHDL と Verilog の利点と不利点を示しています。

VHDL と Verilog の利点と不利点

VHDL	Verilog
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。	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)
HDL ソース コードでの RAM の初期化が簡単 (Verilog の初期ブロックの方が困難)	C 言語のような構文
パッケージ サポート	コードは VHDL よりもコンパクト
カスタム タイプ	コメントのブロック化 (複数行)
列挙型	VHDL のようにコンポーネント インスタンス化が多くない
reg と wire を混乱することがない	

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
- ・ 複雑なマクロはアドバンス HDL 合成中に推論
- ・ マクロの中には、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るものもあり

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。

たとえば、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、推論されることがあります。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 エレメント間の関係が 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う。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

このセクションには、次の内容が含まれます。

- ・ [フリップフロップおよびレジスタの概要](#)
- ・ [フリップフロップおよびレジスタの初期化](#)
- ・ [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関連制約](#)
- ・ [フリップフロップおよびレジスタのレポート](#)
- ・ [フリップフロップおよびレジスタのコード例](#)

フリップフロップおよびレジスタの概要

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

フリップフロップとレジスタは VHDL の順次 process または Verilog の always ブロックで記述されます。

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 言語のサポート」](#)
- ・ [第 4 章「Verilog 言語のサポート」](#)

process または always ブロックのセンシティブティリストには、クロック信号とすべての非同期制御信号がリストされます。

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 でこれを実行するには、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 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

次が制御信号です。

- ・ 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

下記のコーディング ガイドラインに従うと、次が可能になります。

- ・ 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 および DSP ブロックなどのデバイス リソースの使用

コーディング ガイドラインは、次のようになります。

- ・ レジスタを非同期にセット/リセットしないで、同期初期化を使用します。これは、ザイリンクス デバイスでは可能ですが、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。
 - ブロック RAM 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。
- ・ 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。Virtex®-6 および Spartan®-6 デバイスから、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。ソフトウェア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。
- ・ 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。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 ロジックを一緒に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。
- ・ ザイリンクス 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 です。アクティブ Low にすると、インバータ ロジックになり、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 \(IOB\)](#)
- ・ [レジスタの複製 \(REGISTER_DUPLICATION\)](#)
- ・ [等価レジスタの削除 \(EQUIVALENT_REGISTER_REMOVAL\)](#)
- ・ [レジスタの自動調整 \(REGISTER_BALANCING\)](#)
- ・ [非同期から同期への変換 \(ASYNC_TO_SYNC\)](#)

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、[第 8 章「FPGA の最適化」](#)の「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。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ついてもレポートに含まれるようになります。

```
=====
*                               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
```

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。これは、Design Summary セクションには、アドバンス HDL 合成と下位レベル合成で処理された数が表示されるからです。これには、DSP ブロックやブロック RAM 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、レジスタのバランス制御などの情報が含まれます。

フリップフロップおよびレジスタのコード例

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。

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

このセクションには、次の内容が含まれます。

- ・ [ラッチの概要](#)
- ・ [ラッチの記述](#)
- ・ [ラッチの関連制約](#)
- ・ [ラッチのレポート](#)
- ・ [ラッチのコード例](#)

ラッチの概要

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、ラッチ出力をモデリングする信号に 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 (IOB)

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。

```
=====
*                               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

```
=====
```

その他のマクロと異なり、XST は警告をここに出力します。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。この警告メッセージは、潜在的な問題を警告するもので、これにより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。

ラッチのコード例

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。

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

このセクションには、次の内容が含まれます。

- ・ [トリステートの概要](#)
- ・ [トリステートのインプリメンテーション](#)
- ・ [トリステートの関連制約](#)
- ・ [トリステートのレポート](#)
- ・ [トリステートのコード例](#)

トリステートの概要

内部バスを駆動する場合も、ザイリンクス デバイスのあるボードの外部バスを駆動する場合も、通常トリステートバッファは信号と if-else 文 (信号が if-else の分岐の 1 つでハイインピーダンスになっている箇所) によってモデリングされます。これは、次のような異なったコードスタイルで記述できます。

- ・ 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、内部バス (BUFT) または回路の外部ピン (OBUFT) 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。

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 \(TRISTATE2LOGIC\)](#)

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。

```
=====
*                               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

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。

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  
    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
```

カウンタおよびアキュムレータ

このセクションには、次の内容が含まれます。

- ・ [カウンタおよびアキュムレータ](#)
- ・ [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の関連制約](#)
- ・ [カウンタおよびアキュムレータのレポート](#)
- ・ [カウンタおよびアキュムレータのコード例](#)

カウンタおよびアキュムレータ

XST には、カウンタおよびアキュムレータの推論機能があり、ユーザーは、コアのファンクション以外に次のような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

アキュムレータとカウンタ (別名: インクリメントまたはデクリメント) では、加算または減算、もしくはその両方のオペランドが異なります。カウンタを記述する場合、次のようにデスティネーションおよび最初のオペランドは信号または変数で、2 番目のオペランドは定数 1 になります。

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

アキュムレータを記述する場合、デスティネーションおよび最初のオペランドは信号または変数で、2 番目のオペランドは次のいずれかです。

- ・ 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とアキュムレータの方向は、アップ、ダウン、またはアップダウンのいずれかになります。アップダウン アキュムレータの場合、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;
```

XST では、符号付きおよび符号なし両方のカウンタとアキュムレータがサポートされています。

整数型の信号で記述する場合も、ビットの配列で記述する場合も、XST では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。HDL 記述で明示的に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潜在的に使用します。mod 演算子を次のように使用すると、特定の値までカウントアップできます。

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できます。

- ・ スライス ロジック
- ・ DSP ブロック リソース

1 つの DSP ブロックにカウンタまたはアキュムレータがフィットする場合、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 はターゲット デバイスで使用可能な DSP ブロック リソースすべてを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。

カウンタおよびアキュムレータの関連制約

- ・ [DSP ブロックの使用 \(USE_DSP48\)](#)
- ・ [DSP 使用率 \(DSP_UTILIZATION_RATIO\)](#)

カウンタおよびアキュムレータのレポート

カウンタおよびアキュムレータは、HDL 合成中に推論されたレジスタと加算器/減算器マクロ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。次のレポートは、それらのレポートを順番に示しています。

```
=====
*                               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
```

カウンタおよびアキュムレータのコード例

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。

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

    port(
        clk : in std_logic;
        rst : in std_logic;
        D    : in std_logic_vector(3 downto 0);
        Q    : out std_logic_vector(3 downto 0));

end accumulators_2;

architecture archi of accumulators_2 is

    signal cnt : std_logic_vector(3 downto 0);

begin

    process (clk)
    begin
        if (clk'event and clk = '1') then
            if (rst = '1') then
                cnt <= "0000";
            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スカウンタの 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);

    input        clk;
    input        load;
    output [3:0] Q;
    reg [3:0] cnt;

    always @(posedge clk)
    begin
        if (load)
            cnt <= 4'b1100;
        else
            cnt <= cnt - 1'b1;
        end

    assign Q = cnt;

endmodule
```

シフトレジスタ

このセクションには、次の内容が含まれます。

- ・ [シフトレジスタの概要](#)
- ・ [シフトレジスタの記述](#)
- ・ [シフトレジスタのインプリメンテーション](#)
- ・ [シフトレジスタの関連制約](#)
- ・ [シフトレジスタのレポート](#)
- ・ [シフトレジスタのコード例](#)

シフトレジスタの概要

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 レジスタには、通常次が含まれます。

- ・ 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

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もでき、SRL タイプの専用プリミティブを最大限に利用して、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ます。ザイリンクスではこのようなロジックを削除して、その代わりに該当する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

次にシフト レジスタのコア機能を記述する一般的な方法を 2 つ示します。

連結演算子を使用した VHDL コード例

コンパクトにするには、連結演算子を使用します。

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

for-loop 文の VHDL コード例

for-loop 文を使用すると、次のようになります。

```
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

このセクションでは、シフト レジスタのインプリメンテーションについて説明します。

- ・ [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

XST では推論されたシフト レジスタを SRL16、SRL16E、SRLC16、SRLC16E、SRLC32E のような SRL タイプのリソースにインプリメントします。

シフトレジスタの長さによって、1 つの SRL タイプのプリミティブにインプリメントされるか、SRLC タイプのプリミティブのカスケード機能が使用されます。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) にインプリメントすることもできます。この方法を使用すると、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。

ただし、ブロック RAM または LUT RAM にインプリメントする場合、XST では「[シフトレジスタの記述](#)」で説明するようにシフトレジスタをインプリメントする決定を下すことができません。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

ブロック RAM の機能が使用されるよにするには、まず read-first 同期モードにします。また、カウンタがアドレス指定可能な空間を順番にスキャンし、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する必要があります。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。この結果、たとえば深さ 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 が使用される場合、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。

RAM の機能の詳細は、「RAM」を参照してください。

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。

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

このようなシフトレジスタは別のレジスタを使用してインプリメントされた最後のステージを使用すると、分散 RAM にもインプリメントできます。たとえば、深さ 128 の遅延線では 127 ワードのアドレス指定可能なデータと最後 1 つのレジスタステージで LUT RAM を使用しています。

RAM の機能の詳細は、「RAM」を参照してください。

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。

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

シフトレジスタの抽出 (SHREG_EXTRACT)

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。次のレポートは、それらのレポートを順番に示しています。

```
=====
* 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

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。

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);  
  
    input    clk, clken, SI;  
    output   SO;  
    reg [7:0] shreg;  
  
    always @(posedge clk)  
    begin  
        if (clken)  
            shreg = {shreg[6:0], SI};  
    end  
  
    assign SO = shreg[7];  
  
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  
//      Concatenation-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);  
  
    input      clk, clken, SI;  
    output     SO;  
    reg [7:0] shreg;  
  
    integer i;  
  
    always @(posedge clk)  
    begin  
        if (clken)  
        begin  
            for (i = 0; i < 7; i = i+1)  
                shreg[i+1] <= shreg[i];  
            shreg[0] <= SI;  
        end  
    end  
  
    assign SO = shreg[7];  
  
endmodule
```

ダイナミック シフト レジスタ

このセクションには、次の内容が含まれます。

- ・ [ダイナミック シフト レジスタの概要](#)
- ・ [ダイナミック シフト レジスタの関連制約](#)
- ・ [ダイナミック シフト レジスタのレポート](#)
- ・ [ダイナミック シフト レジスタのコード例](#)

ダイナミック シフト レジスタの概要

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られることができるシフトレジスタです。回路操作中に使用される最大長を考慮すると、シフトレジスタはその長さのフリップフロップのチェーンであり、マルチプレクサ選択を使用し、指定されたクロック サイクルで、ステージ データはその伝搬チェーンから抽出されます。次の図は、このコンセプトをまとめたものです。

ダイナミック シフト レジスタ

XST ではすべての最大長のダイナミック シフトレジスタを推論し、ターゲット デバイス ファミリで使用可能な SRL タイプのプリミティブを使用してそれらを最適にインプリメントできます。

ダイナミック シフト レジスタの関連制約

[シフトレジスタの抽出 \(SHREG_EXTRACT\)](#)

ダイナミック シフト レジスタのレポート

HDL 合成中、XST は最初にフリップフロップとマルチプレクサを識別します。ダイナミック シフトレジスタは実際にはアドバンス HDL 合成で認識され、それらの基本マクロ間の依存性が決定されます。次のレポートは、それらのレポートを順番に示しています。

```
=====
* 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

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。

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 example (CLK, CE, A, SI, DO);

    input        CLK, CE, SI;
    input  [4:0] A;
    output       DO;

    reg [31:0] data;

    assign DO = data[A];

    always @(posedge CLK)
    begin
        if (CE == 1'b1)
            data <= {data[30:0], SI};
    end

endmodule
```

マルチプレクサ

このセクションには、次の内容が含まれます。

- ・ [マルチプレクサの概要](#)
- ・ [マルチプレクサのインプリメンテーション](#)
- ・ [マルチプレクサの Verilog の \[Case Implementation Style\] パラメータ](#)
- ・ [マルチプレクサの関連制約](#)
- ・ [マルチプレクサのレポート](#)
- ・ [マルチプレクサのコード例](#)

マルチプレクサの概要

マルチプレクサ マクロは、組み合わせプロセスまたは always ブロックで記述された連結代入文か、順次プロセスまたは always ブロック内など、別のコード スタイルから推論できます。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、セレクトの値がすべて列挙されているか、または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トの値に対して選択されているかを定義しておく必要があります。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。同様に、マルチプレクサが if-elseif 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ト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ト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。MUX を強制的に推論させる場合は、[MUX の抽出 \(MUX_EXTRACT\)](#) 制約を使用します。

マルチプレクサの Verilog の [Case Implementation Style] パラメータ

[Case Implementation Style] パラメータを使用すると、記述した case 文をさらに特性化することができます。

詳細は、[第 9 章「デザイン制約」](#)を参照してください。

[Case Implementation Style] パラメータには、次のいずれかを選択します。

- ・ none (デフォルト)
case 文のビヘイビアを記述どおりにインプリメントします。
- ・ **full**
case 文が完了していると認識され、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**
case 文の分岐は同時に発生しないと判断され、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、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。

[Case Implementation Style] パラメータが実際に使用されると、情報メッセージが表示されます。case 文自体の特性からその必要がない場合 (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 パラメータを full にした場合など)、メッセージは表示されません。

[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と異なるビヘイビアがインプリメントされる場合があります。

マルチプレクサの関連制約

- ・ [MUX の抽出 \(MUX_EXTRACT\)](#)
- ・ [列挙型エンコード手法 \(ENUM_ENCODING\)](#)

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。

```
=====
*                               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
=====
```

マルチプレクサの明示的推論およびレポートは、ターゲット デバイスと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 Virtex®-6 または Spartan®-6 デバイスの場合レポートされません。これらのデバイスの場合、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。

マルチプレクサのコード例

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。

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

このセクションには、次の内容が含まれます。

- ・ [四則演算ブロックの概要](#)
- ・ [XST での四則演算の符号サポート](#)
- ・ [四則演算のインプリメンテーション](#)

四則演算ブロックの概要

XST では、次の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、アキュムレータや乗加算器、DSP フィルタなどのより複雑なブロックを構築できます。

XST での四則演算の符号サポート

このセクションには、次の内容が含まれます。

- ・ [XST での四則演算の符号サポート](#)
- ・ [Verilog の符号サポート](#)
- ・ [VHDL の符号サポート](#)

XST での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

XST で Verilog または VHDL を使用する場合、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 では次の規則が適用されます。

- ・ ベクタ型の port、wire、reg は符号付きと明示されない限り、符号なし (unsigned) として処理されます。
- ・ 整数の変数は、指定されていない限り符号付き (signed) として処理されます。
- ・ 10 進数は符号付きです。
- ・ 基数は、指定されていない限り符号なし (unsigned) です。

データ型の記述を明示的に指定するには、unsigned と signed キーワードを使用します。

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、基数も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

また、\$signed および \$unsigned 変換関数を使用しても、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);
```

Verilog の場合、論理式のタイプはその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。論理式のタイプのレゾリューションは、次の規則に基づいて実行されます。

- ・ ビット セレクトの結果は、オペランドに関係なく符号なし
- ・ パート セレクトの結果は、パート セレクトでベクタ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
- ・ 連結の結果は、オペランドに関係なく符号なし
- ・ 比較の結果は、オペランドに関係なく符号なし
- ・ 自ら決定されるオペランドの符号とビット長は、オペランド自体で決定されるので、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が使用される場合は、Verilog LRM のガイドラインを参照してください。

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。たとえば、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。

符号なしの演算

パッケージ	タイプ
<code>numeric_std</code>	<code>unsigned</code>
<code>std_logic_arith</code>	<code>unsigned</code>
<code>std_logic_unsigned</code>	<code>std_logic_vector</code>

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を処理する演算パッケージおよびタイプを使用できます。

符号付き演算

パッケージ	タイプ
<code>numeric_std</code>	<code>signed</code>
<code>std_logic_arith</code>	<code>signed</code>
<code>std_logic_signed</code>	<code>std_logic_vector</code>

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。

四則演算のインプリメンテーション

このセクションでは、演算のインプリメンテーションについて説明します。次の内容が含まれます。

- ・ [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 では特に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 および Spartan®-6 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。使用可能な DSP ブロックの量は、ターゲット デバイスによって異なり、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可能な限り利用して、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ようとしています。

DSP ブロックへのインプリメンテーションの詳細については、[乗算器と乗加算および乗累算](#)を参照してください。

- ・ 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 \(USE_DSP48\)](#) 制約をデフォルト値の 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、ターゲット 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、デバイスで使用可能な 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。
- ・ スタンドアロンの加算器、アキュムレータ、カウンタなど、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。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。
- ・ 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、その利点が生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。これらのレジスタは同時にリセットできるようにすることもできます。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がされないの、使用しないでください。
- ・ 符号なし (unsigned) の演算を記述する場合、DSP ブロック リソースはオペランドを符号付きとみなすことを覚えておってください。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。たとえば、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。

DSP ブロック リソースの詳細および最適な HDL コーディング方法については、次を参照してください。

- ・ 『Virtex-6 FPGA DSP48E1 Slice User Guide』
(http://japan.xilinx.com/support/documentation/user_guides/ug369.pdf)
- ・ 『Spartan-6 FPGA DSP48A1 Slice User Guide』
(http://japan.xilinx.com/support/documentation/user_guides/ug389.pdf)

コンパレータ

このセクションには、次の内容が含まれます。

- ・ [コンパレータの概要](#)
- ・ [コンパレータの関連制約](#)
- ・ [コンパレータのレポート](#)
- ・ [コンパレータのコード例](#)

コンパレータの概要

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
- ・ 小なり
- ・ 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。

```
=====
*                               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

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。

符号なし 8 ビットの大なりおよび等価コンパレータの 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
    port(A,B : in  std_logic_vector(7 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 ビットの小なりコンパレータの 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);

    input  [7:0] A;
    input  [7:0] B;
    output CMP;

    assign CMP = (A < B) ? 1'b1 : 1'b0;

endmodule
```

除算器

このセクションには、次の内容が含まれます。

- ・ [除算器の概要](#)
- ・ [除算器の関連制約](#)
- ・ [除算器のレポート](#)
- ・ [除算器のコード例](#)

除算器の概要

除算器は、次の場合にのみサポートされます。

- ・ 除数が定数と 2 のべき乗の場合 (このような記述はシフタ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

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。

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 / 3;

endmodule
```

加算器、減算器、加減算器

このセクションには、次の内容が含まれます。

- ・ [加算器、減算器、加減算器の概要](#)
- ・ [キャリー出力の記述](#)
- ・ [加算器、減算器、加減算器のインプリメンテーション](#)
- ・ [加算器、減算器、加減算器の関連制約](#)
- ・ [加算器、減算器、加減算器のレポート](#)
- ・ [加算器、減算器、加減算器のコード例](#)

加算器、減算器、加減算器の概要

XST では、加算器、減算器、加減算器が認識されます。加算器はオプションのキャリー入力とキャリー出力を付けて記述できます。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。

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];
```

キャリー出力を記述した VHDL コード例 2

VHDL で加算器をキャリー出力付きで記述する場合、使用する演算パッケージに注意してください。たとえば、上記方法で `std_logic_unsigned` を使用するのはいちいち適切ではありません。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。

```
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];
```

オペランドは **integer** 型に変換したり、加算結果を次のように **std_logic_vector** に変換したりすることもできます。conv_std_logic_vector 変換関数は、std_logic_arith パッケージに含まれています。符号なしの + 演算は、std_logic_unsigned に含まれています。

```
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、加減算器はデフォルトでは DSP ブロックリソースにインプリメントされず、キャリーロジックを使用して合成されます。

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、出力レジスタの 1 つのレベルの DSP48 ブロックへのインプリメント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。auto モードにすると、加減算器はフィルタのようなより複雑なマクロの一部になり、XST ではこれを自動的に DSP ブロックに配置します。これ以外のモードでは、LUT を使用して加減算器が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。接続がある場合、高速の DSP 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。

XST では、DSP48 に最大数のレジスタを含めるなど、最大限のマクロ コンフィギュレーションを推論およびインプリメントすることで、最良のパフォーマンスを達成しようとします。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 \(KEEP\)](#) 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、[キープ \(KEEP\)](#) 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 \(USE_DSP48\)](#)
- ・ [DSP 使用率 \(DSP_UTILIZATION_RATIO\)](#)
- ・ [キープ \(KEEP\)](#)

加算器、減算器、加減算器のレポート

```
=====
*                               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
```

```
=====
```

キャリー入力付きの加算器の場合、2 つの別々の加算器マクロが最初に推論され、HDL 合成 (HDL Synthesis 部分) でレポートされます。これらは、レポートでアドバンス 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、それがアドバンス HDL 合成レポートに表示されます。同様に、ボロー入力付きの減算器を記述した場合、2 つの別々の減算器マクロが最初に推論され、アドバンス HDL 合成でまとめられます。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、加減算器のコード例

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。

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
    port(A,B : in std_logic_vector(7 downto 0);
          SUM : out std_logic_vector(7 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);
    input [7:0] A;
    input [7:0] B;
    input CI;
    output [7:0] SUM;

    assign SUM = A + B + CI;

endmodule
```


乗算器

このセクションには、次の内容が含まれます。

- ・ [乗算器の概要](#)
- ・ [乗算器のインプリメンテーション](#)
- ・ [乗算器の関連制約](#)
- ・ [乗算器のレポート](#)
- ・ [乗算器のコード例](#)

乗算器の概要

XST では、HDL ソース コードで検出された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。積の全 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

このセクションでは、次の内容について説明します。

- ・ [乗算器のインプリメンテーションの概要](#)
- ・ [DSP ブロックのインプリメンテーション](#)
- ・ [スライス ロジックのインプリメンテーション](#)
- ・ [定数への乗算](#)

乗算器のインプリメンテーションの概要

乗算器マクロは、次のデバイス リソース タイプ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 \(USE_DSP48\)](#) 制約をデフォルト値の 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。

- ・ 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 ファミリによって異なります。DSP ブロックにインプリメントするには、制約の値を yes に指定します。
- ・ 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、ターゲット 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、デバイスで使用可能な 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 \(USE_DSP48\)](#) を no (スライス ロジック) または yes (DSP ブロック) に設定します。

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。

- ・ 乗算オペランドの 2 レベルのレジスタ
- ・ 乗算の後ろの 2 レベルのレジスタ

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、複数の DSP ブロックか、DSP ブロックとスライス ロジックの両方にインプリメントします。どちらの方法になるかは、最大のパフォーマンスになることを目標に、オペランドのサイズによって決まります。

複数の DSP ブロックへのインプリメンテーションは、XST がパイプライン化を実行するように命令されていると、さらに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 は自動的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。使用可能な場合、XST では目標を達成するためにそのレジスタを移動します。レイテンシが十分ではない場合は、アドバンス HDL 合成中に次のような HDL Advisor のメッセージが表示されます。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。

```
INFO:Xst:2385 - HDL ADVISOR - You can improve the performance of the
multiplier Mmult_n0005 by adding 2 register level(s).
```

[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 \(KEEP\)](#) 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、1 ステージまたは複数ステージのレイテンシが使用可能で、ターゲット デバイスで使用可能な DSP ブロック数の制限内である場合、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。乗算器をスライス ロジ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 no に指定します。

乗算器をスライス ロジックにインプリメントする場合、XST は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て、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。このため、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。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。

- ・ KCM (Constant Coefficient Multiplier) インプリメンテーション
- ・ CSD (Canonical Signed Digit) インプリメンテーション

これらの方法を使用した場合の最適化レベルは、その定数オペランドの特徴によって異なります。場合によっては、KCM 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。このため、XST では KCM か通常の乗算インプリメンテーションが自動的に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 \(MULT_STYLE\)](#) を次の目的で使用します。

- ・ CSD インプリメンテーションの指定
- ・ KCM インプリメンテーションの指定

XST では次の場合、KCM 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 \(USE_DSP48\)](#)
- ・ [DSP 使用率 \(DSP_UTILIZATION_RATIO\)](#)
- ・ [キープ \(KEEP\)](#)
- ・ [乗算器スタイル \(MULT_STYLE\)](#)

乗算器のレポート

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、次のように HDL 合成レポートに表示されます。

```
=====
*                               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

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。

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
    port(A : in std_logic_vector(7 downto 0);
         B : in std_logic_vector(3 downto 0);
         RES : out std_logic_vector(11 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);

    input          clk;
    input          [31:0] A;
    input          [23:0] B;
    output         [55:0] RES;

    reg            [31:0] rA;
    reg            [23:0] rB;
    reg            [55:0] 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];
    endmodule
```

乗加算および乗累算

このセクションには、次の内容が含まれます。

- ・ [乗加算および乗累算](#)
- ・ [乗加算と乗累算のインプリメンテーション](#)
- ・ [乗加算と乗累算の関連制約](#)
- ・ [乗加算と乗累算のレポート](#)
- ・ [乗加算と乗累算のコード例](#)

乗加算および乗累算

乗加算、乗減算、乗加減算、乗累算マクロは、乗算器、加減算器および前の HDL 合成段階で推論されたレジスタなどを集めて、アドバンス 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

推論された乗加算または乗累算マクロはザイリンクス デバイスで使用可能な DSP ブロック リソースにインプリメントできます。この場合、XST は次のような DSP ブロックのパイプライン機能を利用しようとしています。

- ・ 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 MAC 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、乗算器とアキュムレータ (累算) が別々のマクロとして処理されます。

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。この auto モードでは、ターゲット デバイスで使用可能な DSP ブロック リソースの量が考慮されて、乗加算および乗累算マクロがインプリメントされます。XST では、デバイスで使用可能な 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。

XST では、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、DSP ブロックのすべてのパイプライン機能を利用して回路パフォーマンスを最大限にしようとしています。[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 \(KEEP\)](#) 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 \(USE_DSP48\)](#)
- ・ [DSP 使用率 \(DSP_UTILIZATION_RATIO\)](#)
- ・ [キープ \(KEEP\)](#)

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、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。

```
=====
*                               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

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。

乗算アップ アキュムレータ (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);

    input        clk;
    input        rst;
    input  [7:0] A, B;
    output [15:0] RES;

    reg  [15:0] mult, accum;
    reg  [7:0] rA, rB;

    always @(posedge clk)
    begin
        if (rst) begin
            rA    <= 8'b00000000;
            rB    <= 8'b00000000;
            mult  <= 16'b0000000000000000;
            accum <= 16'b0000000000000000;
        end
    else begin
        rA <= A;
        rB <= B;

        mult <= rA * rB;
        accum <= accum + mult;
    end
    end

    assign RES = accum;

endmodule
```

DSP の推論

このセクションでは、次の内容について説明します。

- ・ [DSP の推論の概要](#)
- ・ [対称フィルタ](#)
- ・ [DSP の推論のコード例](#)

DSP の推論の概要

XST には、レイテンシ ステージ (レジスタ)、乗算、乗加算/減算、累算、乗累算、ROM などの基本的なファンクションのより精密な推論機能に加え、ポータブルのビヘイビア ソース コードを含むフィルタを記述するための、さらに拡張された推論機能も含まれます。

XST は、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し、ザイリンクス デバイスで使用可能な DSP ブロックリソースの優れた機能 (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) を利用することで、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します。

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 ブロックの詳細と使用方法については、次を参照してください。

- 『Virtex-6 FPGA DSP48E1 Slice User Guide』
(http://japan.xilinx.com/support/documentation/user_guides/ug369.pdf)
- 『Spartan-6 FPGA DSP48A1 Slice User Guide』
(http://japan.xilinx.com/support/documentation/user_guides/ug389.pdf)

対称フィルタ

ザイリンクス DSP ブロックのオプションの前置加算器は、対称フィルタ用に設計されています。対称係数フィルタを記述する場合は、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。

XST では自動的に対称係数が識別されて因数分解されないの、フィルタを一般的な方法で記述するのはお勧めしません。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 コードは、その具体例です。

DSP の推論のコード例

DSP 推論のコード例は、ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip

の HDL_Coding_Techniques/dsp ディレクトリにあります。各デザインは、それぞれの下位ディレクトリに保存されています。

DSP リファレンス デザイン

デザイン	言語	説明	デバイス
PolyDecFilter	VHDL	多層間引きフィルタ	Spartan®-6 Virtex®-6
PolyIntrpFilter	VHDL	多層補間フィルタ	Spartan-6 Virtex-6
EvenSymSystFIR	VHDL	偶数タップ付き対称シストリックフィルタ。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
OddSymSystFIR	VHDL	奇数タップ付き対称シストリックフィルタ。DSP ブロックの前置加算器の利点を生かす	Virtex-6

デザイン	言語	説明	デバイス
		ために、対称係数が因数分解されています。	
EvenSymTranspConvFIR	VHDL	偶数タップ付き対称転置型たたみ込みフィルタ。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
OddSymTranspConvFIR	VHDL	奇数タップ付き対称転置型たたみ込みフィルタ。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
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

リソース共有

このセクションには、次の内容が含まれます。

- ・ [リソース共有の概要](#)
- ・ [リソース共有の関連制約](#)
- ・ [リソース共有のレポート](#)
- ・ [リソース共有のコード例](#)

リソース共有の概要

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。これらの最適化は、四則演算子の数を最小限に抑えて、デバイス使用率を削減することを目的に行われます。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。

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。回路パフォーマンスを主な最適化目標としていて、タイミング目標を達成できない場合、リソース共有をディスエーブルにしておくと、タイミング目標が達成できることがあります。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 \(RESOURCE_SHARING\)](#)

リソース共有のレポート

演算のリソース共有は HDL 合成中に実行され、レポートには次のような演算マクロの統計や特定の HDL Advisor メッセージなどが含まれます。

```
=====
*                               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
```

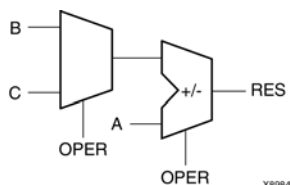
```
=====
INFO:Xst:1767 - HDL ADVISOR - Resource sharing has identified that some
arithmetic operations in this design can share the same physical
resources for reduced device utilization.
For improved clock frequency you may try to disable resource sharing.
```

リソース共有のコード例

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。

この VHDL および Verilog の例では、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

このセクションには、次の内容が含まれます。

- ・ [RAM の概要](#)
- ・ [分散 RAM とブロック RAM](#)
- ・ [RAM のサポート機能](#)
- ・ [RAM の HDL コード記述のガイドライン](#)
- ・ [ブロック RAM の最適化ストラテジ](#)
- ・ [分散 RAM のパイプライン](#)
- ・ [RAM の関連制約](#)
- ・ [RAM のレポート](#)
- ・ [RAM のコード例](#)

RAM の概要

XST の拡張 RAM 推論機能を使用すると、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がありません。また、時間が節約でき、HDL ソース コードを移動および拡張しやすくなります。

分散 RAM とブロック RAM

Virtex®-6 および Spartan®-6 デバイスの RAM リソースには、次の 2 タイプがあります。

- ・ 分散 RAM (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る)
- ・ 専用ブロック RAM リソース

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。

- ・ 分散 RAM の場合は非同期
- ・ ブロック RAM の場合は同期

XST では、どちらのタイプのリソースでも利点を生かすことができます。インプリメンテーションの選択は、次の条件によって決まります。

- ・ HDL で記述した通りの特徴の RAM
- ・ 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
- ・ ターゲット デバイスで使用可能なブロック RAM リソース

ただし、上記の主な違いを考慮すると、次のようになります。

- ・ 非同期読み出しの付いた RAM 記述は、分散 RAM リソースを使用してインプリメントされる必要があります。ブロック RAM にはインプリメントできません。
- ・ 同期読み出しの付いた RAM 記述は、通常ブロック RAM になりますが、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 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。

Virtex-6 および Spartan-6 デバイスの RAM リソースに関する詳細は、次を参照してください。

- ・ [『Virtex-6 FPGA Memory Resources User Guide』](#)
- ・ [『Virtex-6 FPGA Configurable Logic Block User Guide』](#) の分散 RAM の部分
- ・ [『Spartan-6 FPGA Block RAM Resources User Guide』](#)
- ・ [『Spartan-6 FPGA Configurable Logic Block User Guide』](#) の分散 RAM の部分

RAM のサポート機能

XST の 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 記述の 1 つまたは複数の RAM プリミティブへのマップが自動的に処理されます。
- ・ 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。

- ・ 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

次のブロック RAM の機能は、サポートされていません。

- ・ パリティ ビット

XST では、一部のブロック RAM プリミティブで使用可能なパリティビットを通常のデータ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が、パリティ制御ロジックを自動的に生成する機能はなく、これらのパリティビット部分を意図した目的のために使用することはできません。

- ・ 非対称ポート付きの RAM

プライマリとデュアル ポート間の非対称の作成は、ザイリンクス ブロック RAM の優れた機能の 1 つです。ただし、このバージョンの XST では、このような機能の推論はサポートされていません。

RAM の HDL コード記述のガイドライン

このセクションでは、RAM の HDL コード記述のガイドラインについて説明します。

- ・ [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

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 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と、XST で拒否されます。このような記述は、シミュレーション ツールで正しく動作しません。

注意! 「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」はすべての基本特性を引き継ぐので、使用の際には変数よりも注意が必要です。次に注意してください。

- ・ 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「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に「shared variable」を使用するのは、XST では使用できますが、お勧めしません。「signal」を使用してください。

Verilog での RAM の記述

```
reg [15:0] RAM [0:255];
```

書き込みアクセスの記述

このセクションには、次の項目が含まれます。

- ・ [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);
```

注意！ VHDL の場合、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。使用すると、XST ではアドレス信号が符号付きで記述されると仮定され、すべての負の値が無視されます。このため、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。

RAM に書き込みポートが 2 つ付いていて、VHDL の「shared variable」を使用して記述されている場合、通常書き込みは、次のように記述されます。

```
process (clk)
begin
    if rising_edge(clk) then
        if we = '1' then
            RAM(conv_integer(addr)) := di;
        end if;
    end if;
end process;
```

Verilog での書き込みアクセスの記述

書き込みアクセスは、次のように記述します。

```
always @ (posedge clk)
begin
    if (we)
        do <= RAM[addr];
    end
end
```

読み出しアクセスの記述

このセクションには、次の項目が含まれます。

- ・ [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));
```

上記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、読み出しが非同期か同期かが決まります。また、その RAM がブロック RAM リソースと分散 RAM リソースのどちらを使用してザイリンクス デバイ스에インプリメントできるのかも決まります。ブロック RAM リソースにインプリメントする場合は、次のように記述します。

```
process (clk)
begin
    do <= RAM( conv_integer(addr));
end process;
```

詳細は、次の「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。

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、次の「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 (または read-through)
新しい内容が即座に読み出せるようになります。
- ・ 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 VHDL 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 VHDL 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 はそれを認識し、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ように記述します。

```
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;
```

バイト幅の書き込みイネーブルのサポート (ブロック RAM)

ブロック RAM リソースで使用可能なバイト幅の書き込みイネーブル機能を使用すると、RAM へのデータ書き込みが高度に制御できます。これにより、アドレス指定されたメモリ ロケーションの 8 ビットのどの部分に書き込めるようにするかを別々に制御できます。

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の概念に抽象化できます。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 では、バイト幅の書き込みイネーブルのブロック RAM 機能を使用できる推論が可能で、シングル ポートの記述もデュアル ポートの記述も使用できます。このリリースでは、バイト幅の書き込みイネーブル機能は、次の 2 つのプロセスで記述されます。

- ・ 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

read-first モードを使用したこの機能は、次のように記述されます。

```
reg [2*DI_WIDTH-1:0] RAM [SIZE-1:0];
reg [DI_WIDTH-1:0] di0, di1;

always @(we or di or addr or RAM)
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

always @(posedge clk)
begin
    do <= RAM[addr];
    RAM[addr]<={di1,di0};
end

The following variant describes a write-first mode instead.
reg [2*DI_WIDTH-1:0] RAM [SIZE-1:0];
reg [DI_WIDTH-1:0] di0, di1;
reg [DI_WIDTH-1:0] do0, do1;

always @(we or di or addr or RAM)
begin
    if (we[1])
        di1 = di[2*DI_WIDTH-1:1*DI_WIDTH];
        do1 = di[2*DI_WIDTH-1:1*DI_WIDTH];
    else begin
        di1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
        do1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
    end
```

```

end

if (we[0])
    di0 <= di[DI_WIDTH-1:0];
    do0 <= di[DI_WIDTH-1:0];
else begin
    di0 <= RAM[addr][DI_WIDTH-1:0];
    do0 <= RAM[addr][DI_WIDTH-1:0];
end
end

always @(posedge clk)
begin
    do <= {do1,do0};
    RAM[addr]<={di1,di0};
end

```

no-charge モードで RAM をコンフィギュレーションする場合は、次のように記述します。

```

reg    [2*DI_WIDTH-1:0] RAM [SIZE-1:0];
reg    [DI_WIDTH-1:0]   di0, di1;
reg    [DI_WIDTH-1:0]   do0, do1;

always @(we or di or addr or RAM)
begin
    if (we[1])
        di1 = di[2*DI_WIDTH-1:1*DI_WIDTH];
    else begin
        di1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
        do1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
    end

    if (we[0])
        di0 <= di[DI_WIDTH-1:0];
    else begin
        di0 <= RAM[addr][DI_WIDTH-1:0];
        do0 <= RAM[addr][DI_WIDTH-1:0];
    end
end

always @(posedge clk)
begin
    do <= {do1,do0};
    RAM[addr]<={di1,di0};
end

```

非対称ポートのサポート

ブロック RAM リソースは、非同期ポートを含むようにコンフィギュレーションできますが、XST では非対称ポートの推論はサポートされていません。

RAM の初期内容

このセクションでは、次について説明します。

- ・ [HDL ソース コードでの初期内容の指定](#)
- ・ [外部データ ファイルでの初期内容の指定](#)

HDL ソース コードでの初期内容の指定

VHDL の場合、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"
);
```

アドレス指定可能なワードをすべて同じ値に初期化する場合は、次のように記述できます。

```
type ram_type is array (0 to 127) of std_logic_vector (15 downto 0);
signal RAM : ram_type := (others => "0000111100110101");
```

RAM のすべてのビット位置を同じ値に初期化する場合は、次のように記述できます。

```
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 2

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することもできます。

```
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

Verilog の場合、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をすべて同じ値に初期化する場合は、次のように記述することもできます。

```
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を初期化することもできます。

```
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。
- ・ 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 配列の行数と初期化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、警告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 進数で記述します。2 進数と 16 進数を混ぜて使用することはできません。
- ・ 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。

次は、8 X 32 ビット RAM を 2 進数値で初期化するファイルの内容の例です。

```
00001111000011110000111100001111
01001010001000001100000010000100
00000000001111100000000001000001
11111101010000011100010000100100
00001111000011110000111100001111
01001010001000001100000010000100
00000000001111100000000001000001
11111101010000011100010000100100
```

VHDL コード例

このデータは VHDL で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

Verilog では、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
```

詳細は、次を参照してください。

- ・ [第 3 章「VHDL 言語のサポート」の「サポートされる VHDL ファイル タイプ」](#)
- ・ [第 5 章「Verilog ビヘイビア記述のサポート」](#)

ブロック RAM の最適化ストラテジ

このセクションには、次の内容が含まれます。

- ・ [ブロック RAM の最適化ストラテジの概要](#)
- ・ [ブロック RAM のパフォーマンス](#)
- ・ [ブロック RAM のデバイス使用](#)
- ・ [ブロック RAM の電力](#)
- ・ [小型の RAM の条件](#)
- ・ [ブロック RAM へのロジックと FSM のマップ](#)
- ・ [ブロック RAM リソースの管理](#)
- ・ [ブロック RAM のパッキング](#)

ブロック RAM の最適化ストラテジの概要

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るので、パフォーマンス、デバイス使用率、電力などの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。このため、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、このような小型の 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 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ないため、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。RAM 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。

ブロック RAM の電力

XST では、RAM の電力消費を抑えることができます。この方法では、[電力削減 \(POWER\)](#) 合成オプションでイネーブルにした最適化セットの一部が使用されます。イネーブルになると、電力削減だけでなく、エリアと速度の最適化も目標にされます。電力削減が主な目的であり、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は、[RAM スタイル \(RAM_STYLE\)](#) を使用し、値を BLOCK_POWER2 に設定します。

小型の RAM の条件

ブロック RAM リソースを節約するために、XST ではブロック RAM に小型のメモリをインプリメントしないようにします。これには、次の点が影響します。

- ・ ターゲットとするデバイス 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。

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	深さ	深さ * 幅
Spartan®-6	>= 127 ワード	> 512 ビット
Virtex®-6	>= 127 ワード	> 512 ビット

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 を強制的にインプリメンテーションするには、[RAM スタイル \(RAM_STYLE\)](#) を使用してください。

ブロック RAM へのロジックと FSM のマップ

RAM の推論機能に加えて、XST では次をブロック RAM リソースにインプリメントするようにも命令できます。

- ・ FSM コンポーネント
詳細は、第 7 章「HDL コーディング手法」の「FSM コンポーネント」を参照してください。
- ・ 汎用ロジック
詳細は、第 8 章「FPGA の最適化」の「ブロック RAM へのロジックのマップ」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、ターゲット 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、デバイスで使用可能な 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。

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。

- ・ インスタンシエートしたブロック RAM
- ・ 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。XST では、その他の推論された RAM のブロック RAM リソースへのインプリメント前に、これらの制約が適用されます。
- ・ BRAM へのロジックのマップ (BRAM_MAP) を使用したロジックまたは FSM のマップの結果のブロック 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。

ほとんどの場合は XST で回避されますが、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。

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。これは、自動 BRAM パッキング (AUTO_BRAM_PACKING) で制御されます。この制約は、デフォルトではオフになっています。

分散 RAM のパイプライン

レイテンシ ステージ数が適切であった場合、XST では分散リソースにインプリメントされた RAM をパイプライン化して、パフォーマンスを向上します。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. 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. 乗算器スタイル (MULT_STYLE) 制約を pipe_distributed に設定します。

XST は自動的に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。使用可能なレジスタの数がそれよりも少ない場合、アドバンス HDL 合成中に必要なステージ数を示す HDL Advisor のメッセージが表示されます。

記述した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 の抽出 (RAM_EXTRACT)
- ・ RAM スタイル (RAM_STYLE)
- ・ ROM の抽出 (ROM_EXTRACT)
- ・ ROM スタイル (ROM_STYLE)
- ・ BRAM 使用率 (BRAM_UTILIZATION_RATIO)
- ・ 自動 BRAM パッキング (AUTO_BRAM_PACKING)

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可能な推論された 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、サイズ、同期信号、制御信号などを含む推論された RAM に関する詳細な情報がレポートされます。次のログ ファイルの例に示すように、RAM の認識は次の 2 段階に分けて行われます。

- ・ HDL 合成段階中に HDL ソース コードにあるメモリ構造が認識されます。
- ・ アドバンス HDL 合成中には、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を考慮して、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。

```
=====
*                               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 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

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。

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
-- Recommended Modelization with a Signal
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/rams_16a.vhd
--
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity rams_16a 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_16a;

architecture syn of rams_16a is
    type ram_type is array (127 downto 0) of std_logic_vector(15 downto 0);
    signal RAM : ram_type;
begin

    process (CLKA)
    begin
        if CLKA'event and CLKA = '1' then
            if ENA = '1' then
                if WEA = '1' then
                    RAM(conv_integer(ADDRA)) <= DIA;
                end if;
                DOA <= RAM(conv_integer(ADDRA));
            end if;
        end if;
    end process;

    process (CLKB)
    begin
        if CLKB'event and CLKB = '1' then
            if ENB = '1' then
                if WEB = '1' then
                    RAM(conv_integer(ADDRB)) <= DIB;
                end if;
                DOB <= RAM(conv_integer(ADDRB));
            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
```

バイト幅書き込みイネーブル (2 バイト) の付いた 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 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.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;
             DI_WIDTH   : integer := 8);

    port (clk : in  std_logic;
          we  : in  std_logic_vector(1 downto 0);
          addr : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
          di   : in  std_logic_vector(2*DI_WIDTH-1 downto 0);
          do   : out std_logic_vector(2*DI_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 (2*DI_WIDTH-1 downto 0);
    signal RAM : ram_type;

    signal di0, di1 : std_logic_vector (DI_WIDTH-1 downto 0);
begin

    process(we, di)
    begin
        if we(1) = '1' then
            di1 <= di(2*DI_WIDTH-1 downto 1*DI_WIDTH);
        else
            di1 <= RAM(conv_integer(addr))(2*DI_WIDTH-1 downto 1*DI_WIDTH);
        end if;

        if we(0) = '1' then
            di0 <= di(DI_WIDTH-1 downto 0);
        else
            di0 <= RAM(conv_integer(addr))(DI_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;
```

バイト幅書き込みイネーブル (2 バイト) の付いた 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 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

endmodule
```

バイト幅書き込みイネーブル (2 バイト) の付いた 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 with Byte-wide Write Enable (2 bytes) in Write-First Mode
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/rams_25.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_25 is
    generic (SIZE      : integer := 512;
            ADDR_WIDTH : integer := 9;
            DI_WIDTH   : integer := 8);

    port (clk : in  std_logic;
          we  : in  std_logic_vector(1 downto 0);
          addr : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
          di   : in  std_logic_vector(2*DI_WIDTH-1 downto 0);
          do   : out std_logic_vector(2*DI_WIDTH-1 downto 0));
end rams_25;

architecture syn of rams_25 is
    type ram_type is array (SIZE-1 downto 0) of std_logic_vector (2*DI_WIDTH-1 downto 0);
    signal RAM : ram_type;

    signal di0, di1 : std_logic_vector (DI_WIDTH-1 downto 0);
    signal do0, do1 : std_logic_vector (DI_WIDTH-1 downto 0);
begin

    process(we, di, addr, RAM)
    begin
        if we(1) = '1' then
            di1 <= di(2*DI_WIDTH-1 downto 1*DI_WIDTH);
            do1 <= di(2*DI_WIDTH-1 downto 1*DI_WIDTH);
        else
            di1 <= RAM(conv_integer(addr))(2*DI_WIDTH-1 downto 1*DI_WIDTH);
            do1 <= RAM(conv_integer(addr))(2*DI_WIDTH-1 downto 1*DI_WIDTH);
        end if;

        if we(0) = '1' then
            di0 <= di(DI_WIDTH-1 downto 0);
            do0 <= di(DI_WIDTH-1 downto 0);
        else
            di0 <= RAM(conv_integer(addr))(DI_WIDTH-1 downto 0);
            do0 <= RAM(conv_integer(addr))(DI_WIDTH-1 downto 0);
        end if;
    end process;
end;
```

```
end process;

process(clk)
begin
    if (clk'event and clk = '1') then
        do <= do1 & do0;
        RAM(conv_integer(addr)) <= di1 & di0;
    end if;
end process;

end syn;
```

バイト幅書き込みイネーブル (2 バイト) の付いた 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 with Byte-wide Write Enable (2 bytes) in Write-First Mode
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/rams/rams_25.v
//
module v_rams_25 (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;
    reg    [DI_WIDTH-1:0] do0, do1;

    always @(we or di or addr or RAM)
    begin
        if (we[1])
            begin
                di1 = di[2*DI_WIDTH-1:1*DI_WIDTH];
                do1 = di[2*DI_WIDTH-1:1*DI_WIDTH];
            end
        else
            begin
                di1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
                do1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
            end
        end
    end
```

```

        end

        if (we[0])
            begin
                di0 <= di[DI_WIDTH-1:0];
                do0 <= di[DI_WIDTH-1:0];
            end
        else
            begin
                di0 <= RAM[addr][DI_WIDTH-1:0];
                do0 <= RAM[addr][DI_WIDTH-1:0];
            end
        end

    end

    always @(posedge clk)
    begin
        do <= {do1,do0};
        RAM[addr]<={di1,di0};
    end

endmodule

```

バイト幅書き込みイネーブル (2 バイト) の付いた No-Change モードのシングル ポート ブロック RAM の VHDL コード例

XST では、HDL 合成中に do1 と do0 信号に対してラッチが推論されます。これらのラッチは、アドバンス HDL 合成段階でブロック RAM に吸収されます。

```

--
-- Single-Port Block RAM with Byte-wide Write Enable (2 bytes) in No-Change Mode
--
-- Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/rams/rams_26.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_26 is
    generic (SIZE      : integer := 512;
            ADDR_WIDTH : integer := 9;
            DI_WIDTH   : integer := 8);

    port (clk : in std_logic;
          we  : in std_logic_vector(1 downto 0);
          addr : in std_logic_vector(ADDR_WIDTH-1 downto 0);
          di   : in std_logic_vector(2*DI_WIDTH-1 downto 0);
          do   : out std_logic_vector(2*DI_WIDTH-1 downto 0));
end rams_26;

```

```

architecture syn of rams_26 is
    type ram_type is array (SIZE-1 downto 0) of std_logic_vector (2*DI_WIDTH-1 downto 0);
    signal RAM : ram_type;

    signal di0, di1 : std_logic_vector (DI_WIDTH-1 downto 0);
    signal do0, do1 : std_logic_vector (DI_WIDTH-1 downto 0);
begin

    process(we, di, addr, RAM)
    begin
        if we(1) = '1' then
            di1 <= di(2*DI_WIDTH-1 downto 1*DI_WIDTH);
        else
            di1 <= RAM(conv_integer(addr))(2*DI_WIDTH-1 downto 1*DI_WIDTH);
            do1 <= RAM(conv_integer(addr))(2*DI_WIDTH-1 downto 1*DI_WIDTH);
        end if;

        if we(0) = '1' then
            di0 <= di(DI_WIDTH-1 downto 0);
        else
            di0 <= RAM(conv_integer(addr))(DI_WIDTH-1 downto 0);
            do0 <= RAM(conv_integer(addr))(DI_WIDTH-1 downto 0);
        end if;
    end process;

    process(clk)
    begin
        if (clk'event and clk = '1') then
            RAM(conv_integer(addr)) <= di1 & di0;
            do <= do1 & do0;
        end if;
    end process;

end syn;

```

バイト幅書き込みイネーブル (2 バイト) の付いた No-Change モードのシングル ポート ブロック RAM の Verilog コード例

XST では、HDL 合成中に do1 と do0 信号に対してラッチが推論されます。これらのラッチは、アドバンス HDL 合成段階でブロック RAM に吸収されます。

```

//
// Single-Port Block RAM with Byte-wide Write Enable (2 bytes) in No-Change Mode
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: HDL_Coding_Techniques/rams/rams_26.v
//
module v_rams_26 (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;
reg    [DI_WIDTH-1:0]  do0, do1;

always @(we or di or addr or RAM)
begin
    if (we[1])
        di1 = di[2*DI_WIDTH-1:1*DI_WIDTH];
    else
        begin
            di1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
            do1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
        end

    if (we[0])
        di0 <= di[DI_WIDTH-1:0];
    else
        begin
            di0 <= RAM[addr][DI_WIDTH-1:0];
            do0 <= RAM[addr][DI_WIDTH-1:0];
        end
    end

end

always @(posedge clk)
begin
    do <= {do1,do0};
    RAM[addr]<={di1,di0};
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_P1681_1984;
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

このセクションには、次の内容が含まれます。

- ・ [ROM の概要](#)
- ・ [ROM の詳細](#)
- ・ [ROM のインプリメンテーション](#)
- ・ [ROM の関連制約](#)
- ・ [ROM のレポート](#)
- ・ [ROM のコード例](#)

ROM の概要

HDL 記述とインプリメンテーションの点では、読み出し専用メモリ (ROM) は RAM と類似しています。正しくレジスタを付けておくと、ROM もブロック RAM リソースにインプリメントできます。

ROM の詳細

このセクションでは、ROM の詳細について説明します。

- ・ [ROM のコード記述](#)
- ・ [読み出しアクセスの記述](#)

ROM のコード記述

ROM は、VHDL で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。このオブジェクトは、定数または信号のいずれかにできます。

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"
);
```

初期ブロックで記述された ROM の Verilog コード例

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ず、各アドレス値を必ず列挙する必要があります。

```
reg [15:0] rom [15:0];

initial begin
    rom[0] = 16'b0011111100000010;
    rom[1] = 16'b0000000100001001;
    rom[2] = 16'b0001000000111000;
    rom[3] = 16'b0000000000000000;
    rom[4] = 16'b1100001010011000;
    rom[5] = 16'b0000000000000000;
    rom[6] = 16'b0000000110000000;
    rom[7] = 16'b011111111110000;
    rom[8] = 16'b0010000010001001;
    rom[9] = 16'b0101010101011000;
    rom[10] = 16'b1111111010101010;
    rom[11] = 16'b0000000000000000;
    rom[12] = 16'b1110000000001000;
    rom[13] = 16'b0000000110001010;
    rom[14] = 16'b0110011100010000;
    rom[15] = 16'b0000100010000000;
end
```

case 文を使用した ROM のコード例

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
```

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柔軟性が増す

詳細は、第 7 章「HDL コーディング手法」の「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。

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、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 \(ROM_STYLE\)](#) 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 \(ROM_STYLE\)](#)

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。

```
=====
*                               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

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。

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
```

FSM コンポーネント

このセクションには、次の内容が含まれます。

- ・ [FSM コンポーネントの概要](#)
- ・ [FSM コンポーネントの説明](#)
- ・ [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 アーキテクチャ サポート](#)
- ・ [FSM 関連の制約](#)
- ・ [FSM レポート](#)
- ・ [FSM コード例](#)

FSM コンポーネントの概要

XST には、同期 FSM (有限ステート マシン) コンポーネント特有の推論機能、複数のビルトイン FSM エンコーディング ステージを使用して、最適化目標を達成しようとする機能があります。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 \(FSM_EXTRACT\)](#) を使用します。

FSM の説明

このセクションには、次の項目が含まれます。

- ・ [FSM の概要](#)
- ・ [ステート レジスタ](#)
- ・ [次ステートの論理式](#)
- ・ [達成不可能ステート](#)
- ・ [Finite State Machine \(FSM\) 出力](#)
- ・ [Finite State Machine \(FSM\) 入力](#)
- ・ [ステート エンコード手法](#)

Finite State Machine (FSM) の概要

XST では、ムーア型とミラー型（ミレー型）の両方の FSM (有限ステート マシン) がサポートされます。

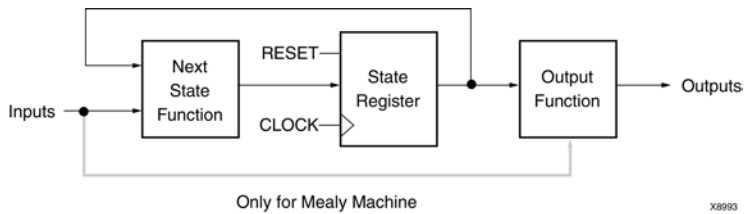
さまざまなコード記述方法がありますが、次のガイドラインを使用すると、可読性が増し、FSM を認識する XST の機能を最大限に生かすことができます。

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

次の図は、ミラー型の FSM を示しています。

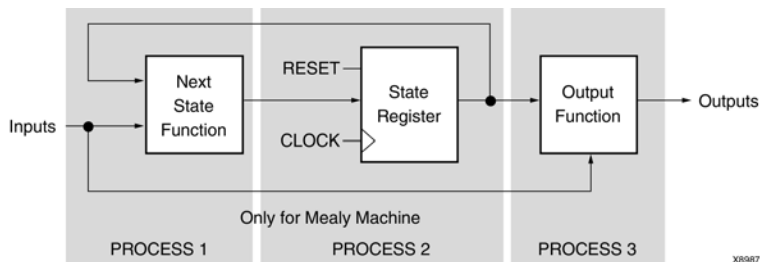
ミーリー マシンおよびムーア マシンを取り入れた FSM の図



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。

- ・ 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。
- ・ ステートレジスタと次ステート関数を一緒に 1 つの順次プロセスまたは always ブロックで記述し、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。
- ・ ステートレジスタを 1 つの順次プロセスまたは always ブロックで記述し、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。
- ・ ステートレジスタを 1 つの順次プロセスまたは always ブロックで記述し、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を記述し、2 つ目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。

3 つのプロセス ブロック使用した 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。

パワーアップ値は、レジスタ パワーアップ (REGISTER_POWERUP) 制約でも指定できます。

ステート レジスタの VHDL コード例

ステートレジスタは、integer、bit_vector、std_logic_vector などの規格タイプを使用して VHDL で指定できます。その他のコード記述方法としては、可能なすべてのステート値を含む列挙型を定義し、そのタイプでステートレジスタを宣言するのが一般的です。

```
type state_type is (state1, state2, state3, state4);
signal state : state_type;
```

ステートレジスタの Verilog コード例

Verilog では、ステートレジスタのタイプに整数または定義されたパラメータのセットを使用できます。

```
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

次ステートの論理式は、順次プロセスで直接記述するか、または別の組み合わせプロセスで記述できます。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。別の組み合わせプロセスを使用する場合は、センシティビティリストにステート信号およびすべての FSM 入力を含める必要があります。

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

レジスタを介さない出力は、組み合わせプロセスまたは同時処理代入文で記述します。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、順次プロセスで代入する内部信号を使用して記述します。

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複数含まれています。[FSM エンコード方法の指定 \(FSM_ENCODING\)](#) で該当するエンコード方法を選択します。

自動ステート エンコード

自動モードの場合、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

ワンホット ステート エンコードは、デフォルトのエンコード手法で、各 FSM ステートにコードの各ビットが代入されます。この結果、ステートレジスタはステートごとにフリップフロップ 1 つを使用してインプリメントされます。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。ワンホット ステート エンコードは、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。

グレイ ステート エンコード

グレイ ステート エンコードは、

- ・ 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

このエンコード手法は、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。この手法はハイパーキューブ イメージョンに基づいており、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

このエンコード手法は、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

この手法では、長いパスを特定し、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が最小化されます。

Speed1 ステート エンコード

Speed1 ステート エンコードは、スピードを最適化する場合に使用します。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 ステート エンコードでは、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。たとえば、ステートレジスタが列挙型で記述される場合に [列挙型エンコード手法 \(ENUM_ENCODING\)](#) 制約を使用すると、各ステートに特定の 2 進数値を割り当て、ユーザー ステート エンコーディングを選択して XST にユーザーのコード手法に従うように命令することができます。詳細は、[第 9 章「デザイン制約」](#)を参照してください。

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。ターゲット デバイスでスライス 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。このようにインプリメンテーションすると、大型の FSM コンポーネントの場合でもパフォーマンスを向上させることができます。デフォルトのスライス ロジックへのインプリメンテーションとブロック RAM へのインプリメンテーションを選択するには、[FSM スタイル \(FSM_STYLE\)](#) を使用します。この制約には、次の値が使用できます。

- ・ **lut** (デフォルト)
- ・ **bram**

XST でブロック RAM への FSM のインプリメント要求が受け入れられない場合は、次のいずれかが実行されます。

- ・ ステート マシンが自動的にスライス 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 セーフ インプリメンテーション

XST では、無効なステートから回復できるようにするロジックを追加して FSM をインプリメントできます。ステート マシンが無効なステートになった場合は、XST で追加されたロジックによってリカバリ ステートと呼ばれる既知のステートに戻されます。このプロセスは、セーフ インプリメンテーション モードと呼ばれます。セーフ FSM のインプリメンテーションをオンにするには、[セーフ インプリメンテーション \(SAFE_IMPLEMENTATION\)](#) を使用します。

XST では、リセット ステートがリカバリ ステートとして自動的に選択されます。どちらのステートも使用できない場合は、パワーアップ ステートが代わりに選択されます。[セーフ リカバリ ステート \(SAFE_RECOVERY_STATE\)](#) 制約を適用すると、手動で特定のリカバリ ステートを定義できます。

FSM 関連の制約

- ・ [FSM 自動抽出 \(FSM_EXTRACT\)](#)
- ・ [FSM スタイル \(FSM_STYLE\)](#)
- ・ [FSM エンコード方法の指定 \(FSM_ENCODING\)](#)
- ・ [列挙型エンコード手法 \(ENUM_ENCODING\)](#)
- ・ [セーフ インプリメンテーション \(SAFE_IMPLEMENTATION\)](#)
- ・ [セーフ リカバリ ステート \(SAFE_RECOVERY_STATE\)](#)

FSM のレポート

XST のログ ファイルには、認識された FSM コンポーネントに関する詳細情報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。

```
=====
*                               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

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。

1 つのプロセス文でステート マシンを記述した 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ステート マシンのコード例

```
//
// 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

このセクションには、次の内容が含まれます。

- ・ [ブラック ボックスの概要](#)
- ・ [ブラック ボックスの関連制約](#)
- ・ [ブラック ボックスのレポート](#)
- ・ [ブラック ボックスのコード例](#)

ブラック ボックスの概要

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
- ・ 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、コードにインスタンスエートする必要があります。これを XST で実行させるには、HDL ソース コードにブラック ボックスのインスタンスエーションを使用します。インスタンス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ンスエーションに制約を指定することも可能です。指定した制約も、NGC ファイルに記述されます。

また、デザイン ブロックの 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、[BoxType \(BOX_TYPE\)](#) 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。

詳細は、次を参照してください。

- ・ [第 10 章「一般制約」](#)
- ・ [『制約ガイド』](#)

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。

コンポーネントのインスタンスエーションの詳細は、VHDL/Verilog のマニュアルを参照してください。

ブラック ボックスの関連制約

ボックス タイプ (BOX_TYPE)

BOX_TYPE は XST でデバイス プリミティブをインスタンスエートするために導入された制約です。この制約を使用する前に、[第 8 章「FPGA プリミティブの最適化」](#)の「[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エーション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 エラボレーションでは、このようなメッセージは表示されません。

ブラック ボックスのコード例

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。

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

メモ: 『XST ユーザー ガイド (Virtex-6 および Spartan-6 用)』は、Virtex®-6 および Spartan®-6 デバイス専用のマニュアルです。その他のデバイスを使用して XST を実行する場合は、『XST ユーザー ガイド』を参照してください。

この章には、次の内容が含まれます。

- ・ 下位レベルの合成
- ・ ブロック RAM へのロジックのマッピング
- ・ 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
- ・ フリップフロップのリタイミング
- ・ エリア制約を設定した場合のスピード最適化
- ・ インプリメンテーション制約
- ・ Xilinx デバイス プリミティブのサポート
- ・ UniMacro ライブラリの使用
- ・ コアの処理
- ・ LUT へのロジックのマッピング
- ・ デバイス上の配置の制御
- ・ バッファの挿入
- ・ XST での PCI™ フローの使用

下位レベルの合成

下位レベル合成の間に、XST のインプリメンテーションを制御するさまざまな方法を使用すると、デザイン目標を達成できます。下位レベルの合成中、XST では次が実行されます。

1. ターゲットにするデバイス ファミリーリソースに対して、各 VHDL エンティティまたは Verilog モジュールを別々にマッピングして最適化します。
2. グローバルに完了したデザインを最適化します。

下位レベル合成の出力は、NGC ネットリスト ファイルです。

XST のデフォルトのインプリメンテーション方法を変更するオプションや制約が複数あります。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。

ブロック RAM へのロジックのマップ

デザインがターゲット デバイスに収まらない場合は、デザインの一部のロジックを未使用のブロック RAM に配置します。XST ではどのロジックをブロック RAM に配置できるかは自動的に決定されないで、ユーザーが指定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 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 [キープ \(KEEP\)](#) 制約を使用できない。

ブロック RAM へのロジックのマップは、下位レベル合成段階で実行されます。問題がなく終了すると、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 にマップされず、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

Virtex®-6 および Spartan®-6 デバイス ファミリーから、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なくなりました。XST では、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、そのフリップフロップが推論されるか、古いデバイス ファミリーのプリミティブ インスタンスレーションからターゲットを変更するかが次のように決定されます。

- ・ 同時同期セットとリセットのターゲットが変更され、追加ロジックが作成される
- ・ 同時非同期セットとリセットが拒否され、次のようなエラー メッセージが表示される

```
ERROR:Xst:#### - This design infers one or more latches or registers
with both an active asynchronous set and reset. In the Virtex6 and
Spartan6 architectures this behaviour 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.
```

次の追加ガイドラインに従ってください。

- ・ レジスタを非同期にセット/リセットしないで、同期初期化を使用します。これは、ザイリンクス デバイスではサポートされますが、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。
 - ブロック RAM および DSP ブロックなどの複数のデバイス 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。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。
- ・ 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/リセットを使用する方法に切り替えることができます。**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、インスタンス化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。Virtex-6 および Spartan-6 デバイスから、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/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。
- ・ ザイリンクス 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 です。アクティブ Low にすると、インバータ ロジックになり、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

このセクションでは、フリップフロップのリタイミングについて説明します。

- ・ [フリップフロップのリタイミングの概要](#)
- ・ [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の制御方法](#)
- ・ [パーティション](#)

フリップフロップのリタイミングの概要

フリップフロップのリタイミングとは、同期パスを削減してクロック周波数を上げるため、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。この最適化は、デフォルトではオフになっています。

フリップフロップのリタイミングには順方向と逆方向があります。

- ・ 順方向のリタイミングでは、LUT の各入力に接続されたフリップフロップすべてを出力で 1 つのフリップフロップに移動します。
- ・ 逆方向のリタイミングでは、LUT の出力にある 1 つのフリップフロップを LUT の各入力に移動します。
- ・ 逆方向のリタイミングを実行すると、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。
- ・ 順方向のリタイミングを実行すると、通常はフリップフロップの数が減ります。

どちらの場合でも、デザインのビヘイビアに変更はなく、タイミング遅延のみ変更されます。

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。リタイミングは反復プロセスであり、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。リタイミングは、指定したタイミング制約が満たされた場合、またはタイミングがそれ以上向上しない場合に停止されます。

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、順方向リタイミングは発生しません。
- ・ フリップフロップの入力信号に **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 の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。

- ・ **レジスタの自動調整 (REGISTER_BALANCING)**
- ・ **最初のフリップフロップ ステージの移動 (MOVE_FIRST_STAGE)**
- ・ **最後のフリップフロップ ステージの移動 (MOVE_LAST_STAGE)**

パーティション

パーティションの詳細は、ISE® Design Suite ヘルプを参照してください。

エリア制約を設定した場合のスピード最適化

XST で主な目標をエリアの削減と指定した場合でも、**スライス (LUT-FF ペア) 使用率 (SLICE_UTILIZATION_RATIO)** 制約を使用すると回路全体のパフォーマンスをある程度制御できます。この制約は、デフォルトでは選択したデバイスサイズの 100% に設定されており、次のように下位レベル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限り、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。

この制約ではマクロ推論は制御されません。

下位レベルの合成のレポート例

次の例では、エリア制約はデバイス サイズの 100% に設定されていますが、最初のエリア概算では実際のデバイス使用率が 102% であることが示され、XST により最適化が実行され、95% に削減されています。

```
=====
* Low Level Synthesis
=====
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.
```

エリア制約を満たすことができない場合、タイミング最適化の際にエリア制約は無視され、周波数が最大になるように下位レベルの合成が実行されます。次の例では、エリア制約が 70% に設定されていますが、XST ではこの制約を満たすことができないため、次のような警告メッセージが表示されます。

```
=====
*                               Low Level Synthesis                               *
=====

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
```

(+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場合、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% 以下であれば、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、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。

次の例では、エリア ターゲットが 55% に設定されていますが、XST では 60% しか達成されていません。ただし、制約と実際のエリアの差が 5% 未満なので、エリア制約は満たされ、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。

```
=====
*                               Low Level Synthesis                               *
=====

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.
```

場合によっては自動リソース管理機能をオフにする必要があります。オフにするには、SLICE_UTILIZATION_RATIO の値を -1 に指定します。[スライス \(LUT-FF ペア\) 使用率 \(SLICE_UTILIZATION_RATIO\)](#) は、デザインの特定ブロックに対して設定できます。スライスの絶対数 (または FF-LUT ペア) またはデバイスの合計スライス数のパーセントを指定できます。

インプリメンテーション制約

HDL ソース コードまたは XCF 制約ファイルで検出されたインプリメンテーション制約はすべて NGC 出力ファイルに書き込まれます。バッファ挿入中には、最大ファンアウトの制御や最適化用に [Keep \(KEEP\)](#) プロパティが生成されます。

ザイリンクス デバイス プリミティブのサポート

このセクションでは、ザイリンクス デバイス プリミティブのサポートについて説明します。

- ・ [ザイリンクス デバイス プリミティブのサポートの概要](#)
- ・ [属性を使用したプリミティブの生成](#)
- ・ [プリミティブとブラック ボックス](#)
- ・ [VHDL および Verilog のザイリンクス デバイス プリミティブ ライブラリの概要](#)
- ・ [プリミティブ プロパティの指定](#)
- ・ [インスタンス化されたデバイス プリミティブのレポート](#)
- ・ [プリミティブの関連制約](#)
- ・ [プリミティブのコード例](#)

デバイス プリミティブのサポートの概要

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化できます。これらのプリミティブは、次のようになります。

- ・ UNISIM ライブラリでコンパイル済み
- ・ デフォルトでは XST で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

[インスタンス化されたプリミティブの最適化 \(OPTIMIZE_PRIMITIVES\)](#) を使用すると、このデフォルト設定を解除でき、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化を簡単にするために、UniMacro という別の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。

- ・ 回路のプライマリ I/O または内部信号に [バッファ タイプ \(BUFFER_TYPE\)](#) を設定すると、特定のバッファ タイプを使用できます。同じ制約を使用してバッファの挿入をディスエーブルにすることもできます。
- ・ [I/O 規格 \(IOSTANDARD\)](#) 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。この例では、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 セーフ イン プリメンテーション](#)』を参照してください。

ブラック ボックスのサポートとプリミティブのサポートは大きく異なります。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。

ボックス タイプ (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 により固有の名前に変更され、警告メッセージが表示されます。この例の場合、MUX5 は MUX51 という名前に変更されています。

```
WARNING:Xst:79 - Model 'muxf5' has different characteristics in destination library
```

```
WARNING:Xst:80 - Model name has been changed to 'muxf51'
```

MUXF5 に **ボックス タイプ (BOX_TYPE)** を設定しない場合、このモジュール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 により固有の名前に変更され、警告メッセージが表示されます。

VHDL および Verilog のザイリンクス デバイス プリミティブ ライブラリ

このセクションでは、次の項目について説明します。

- ・ **VHDL および Verilog のザイリンクス デバイス プリミティブ ライブラリの概要**
- ・ **VHDL および Verilog のザイリンクス デバイス プリミティブ ライブラリ**
- ・ **Verilog のザイリンクス デバイス プリミティブ ライブラリ**
- ・ **プリミティブ インスタンスのガイドライン**

VHDL および Verilog のザイリンクス デバイス プリミティブ ライブラリの概要

VHDL および Verilog のザイリンクス デバイス プリミティブ ライブラリの概要

XST では、HDL コードでザイリンクス デバイス プリミティブのインスタンスをシンプルにするため、VHDL および Verilog の専用ライブラリが提供されています。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、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適切に含めた場合は、**ボックス タイプ (BOX_TYPE)** を適用する必要はありません。

VHDL のザイリンクス デバイス プリミティブ ライブラリ

VHDL の場合、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;
```

このパッケージの 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 のザイリンクス デバイス プリミティブ 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。このライブラリは自動的にデザインにリンクされません。

プリミティブ インスタンスレーションのガイドライン

プリミティブをインスタンスエートする際は、ジェネリック (VHDL) およびパラメータ (Verilog) に大文字を使用してください。たとえば、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エートする場合、DDR_CLK_EDGE および SRTYPE ジェネリック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。LUT1 のようなプリミティブでは、インスタンスエーションで 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エートしたプリミティブに設定します。
- ・ VHDL のジェネリックまたは Verilog のパラメータを使用して渡します。これにより、合成とシミュレーションで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

インスタンスエートされた LUT の INIT のように、VHDL のジェネリックまたは Verilog のパラメータを使用してインスタンスエートされたプリミティブにプロパティを指定します。この際、潜在的なシミュレーション不一致を避けるため、HDL ソースコードや XCF 制約ファイルで属性を指定するのはお勧めしません。この方法で属性を指定すると、次のような警告メッセージが表示されます。

```
WARNING:Xst:#### - Instance <I_FDE> has both an attribute 'INIT'
(value: 0) and a generic/parameter with the same name (value: 1),
potentially leading to a simulation mismatch. VHDL Generics or Verilog
parameters are the generally recommended mechanism to attach properties
to instantiated device primitives or black-boxes.
```

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が認識され、回路の有効化プロセスを簡素化します。

インスタンスエートされたデバイス プリミティブのレポート

インスタンスエートされたプリミティブは、**ボックス タイプ (BOX_TYPE)** とその値が 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 で警告メッセージが表示されるのは、次の場合のみです。

- ・ ブロック (プリミティブ以外) をインスタンス化し、さらに
 - ・ そのブロックに内容がない場合 (ロジック記述なし)
- または
- ・ ブロックにロジック記述があり、さらに
 - ・ [ボックス タイプ \(BOX_TYPE\)](#) を user_black_box で適用した場合

```
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 \(BOX_TYPE\)](#)
- ・ XST で特別な処理をせずに HDL から NGC に渡される配置配線の制約

プリミティブのコード例

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。

LUT2 プリミティブをジェネリック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 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 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、RAM のような複雑なプリミティブのインスタンス化を簡単にするために、UniMacro という別の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。

VHDL の場合、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、ザイリンクス ソフトウェアのインストール ディレクトリにある vhdl¥src¥unisims¥unisims_vcomp.vhd ファイルに含まれています。

Verilog の場合、UniMacro ライブラリがあらかじめコンパイルされています。このライブラリは自動的にデザインにリンクされません。

コアの処理

このセクションでは、次について説明します。

- ・ [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

デザインに EDIF または NGC ネットリスト ファイル形式で記述されたコアが含まれる場合、それらのファイルは自動的に読み込まれ、タイミングの概算およびリソース使用率の制御に使用されます。

この機能をオン/オフにするには、

- ・ ISE® Design Suite の [Process Properties] ダイアログ ボックスを表示して、[Synthesis Options] → [Read Cores] を選択します。
- ・ コマンド ライン モードの場合は、-read_cores を使用します。

この場合、最適化の値を追加して、XST でコアのネットリストがデザイン全体に統合されて、最適化されるようにできます。XST では、デフォルトでコアが読み込まれます。

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 の [Process Properties] ダイアログ ボックスを表示して、[Synthesis Options] → [Core Search Directories] を選択します。
- ・ コマンドライン モードの場合は、-sd を使用します。

コアがあるディレクトリはこれらの方法で指定し、その情報は最新に保つようにしてください。これにより、より良いタイミングやリソース概算が算出されるだけでなく、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ことができます。

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

```
=====
*                               Low Level Synthesis                               *
=====
Launcher: Executing edif2ngd -noa "my_add.edn" "my_add.ngo"
INFO:NgdBuild - Release 11.2 - edif2ngd
INFO:NgdBuild - Copyright (c) 1995-2009 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、UUNISIM 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 関数を計算するのが便利ではないこともあります。この場合は、ほかの方法を使用できます。たとえば、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する方法があります。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。LUT の INIT 値は XST により自動的に算出され、最適化中この LUT が保持されます。詳細は、[「単一 LUT へのエンティティのマッピング \(LUT_MAP\)」](#)を参照してください。

XST では、Synplicity の XC_MAP 制約が自動的に認識されます。

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 xcmap entity <v_and_one> in one lut.
```

LUT へのロジック マッピングの Verilog コード例

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。

次の例では、top ブロックが and_one および and_two で記述される 2 つの AND ゲートをインスタンス化しています。このコードを合成すると 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、ターゲット 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

指定して配置するには、次のコード例のようにレジスタを表す信号または RAM に **RLOC** を適用します。レジスタにこの制約を使用した場合、この制約が各フリップフロップに分配され、**RLOC** 制約は最終ネットリストに渡されます。**RLOC** は、1 つのブロック RAM プリミティブを使用してインプリメント可能な推論済み RAM でサポートされます。

4 ビット レジスタの RLOC 制約の VHDL コード例

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。

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 バッファの挿入は [I/O バッファの追加 \(-iobuf\)](#) でイネーブルまたはディスエーブルにできます。デフォルトではイネーブルになっています。

クロックと I/O バッファは手動でインスタンスエートすることもできます。XST ではインスタンスエートされたデバイス プリミティブを変更しようとせず、単に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

このセクションでは、XST で PCI™ フローを使用する方法について説明します。

- ・ [XST での PCI フローの使用の概要](#)
- ・ [ロジックとフリップフロップの複製の回避](#)
- ・ [コアの自動読み込み機能のオフ](#)

XST での PCI フローの使用の概要

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します。

- ・ VHDL デザインでは、生成されたネットリスト内の名前をすべて大文字にする必要があります。デフォルトでは、小文字になっています。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]

- ・ Verilog デザインでは、[Case] が [Maintain] に設定されていることを確認してください。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]

- ・ デザインの階層を保持します。 [階層の維持 \(KEEP_HIERARCHY\)](#) は、ISE Design Suite で次から指定します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Keep Hierarchy]

- ・ 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。 [等価レジスタの削除 \(EQUIVALENT_REGISTER_REMOVAL\)](#) は、ISE Design Suite で次から指定します。

[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、次のいずれかの操作を行います。

- ・ ISE® Design Suite で [Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスを表示し、[Xilinx Specific Options] ページの [Max Fanout] 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。

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 (例: `max_fanout=2048`)。

コアの自動読み込み機能のオフ

[Read Cores] (READ_CORES) をオフにすると、タイミングおよびエリア予測のため PCI™ コアが自動的に読み込まれないようにできます。PCI コアが読み込まれる場合は、ロジック最適化が実行され、タイミング要件を満たさなかったり、マップ中にエラーになるようなロジックに対して最適化が実行されます。[Read Cores] (READ_CORES) をオフにするのは、[Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスを表示し、[Synthesis Options] ページの [Read Cores] で設定できます。

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。

デザイン制約

メモ: 『XST ユーザー ガイド (Virtex-6 および Spartan-6 用)』は、Virtex®-6 および Spartan®-6 デバイス専用のマニュアルです。その他のデバイスを使用して XST を実行する場合は、『XST ユーザー ガイド』を参照してください。

この章では、XST デザイン制約の一般的な情報について説明します。

- ・ [制約の概要](#)
- ・ [制約の指定](#)
- ・ [制約の優先順序](#)
- ・ [ISE® Design Suite の合成オプション](#)
- ・ [VHDL 属性](#)
- ・ [Verilog-2001 の属性](#)
- ・ [XST Constraint File \(XCF\)](#)

特定の XST デザイン制約の詳細は、次の章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 \(タイミング制約以外\)」](#)
- ・ [第 13 章「タイミング制約」](#)
- ・ [第 14 章「サポートされるサードパーティ制約」](#)

制約の概要

制約は、デザインの目標を達成したり、最適なインプリメンテーションを実行するのに役立ちます。制約を使用すると、合成プロセスや配置配線のさまざまな処理を制御できます。デフォルトの合成アルゴリズムや経験則は、さまざまなデザインで最適な結果になるように設定されていますが、合成の結果に満足がいかない場合は、制約を使用してその他デフォルト以外の設定で合成し直すことができます。

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。

- ・ オプションを使用して合成をグローバルに制御します。これらは ISE® Design Suite かコマンドラインモードの run コマンドから設定できます。詳細は、第 2 章「XST プロジェクトの作成および合成」の「コマンドラインモードからの XST の実行」を参照してください。
- ・ VHDL 属性を直接 VHDL コードに挿入し、デザインの各エレメントに設定して合成および配置配線を制御します。
- ・ 制約は Verilog 属性 (推奨) またはメタコメントとして追加できます。
- ・ 制約を別の制約ファイルで指定することもできます。

通常、グローバルな合成オプションは、ISE Design Suite または XST コマンドラインで定義します。VHDL/Verilog 属性や Verilog メタコメントは HDL ソースコードに挿入して、デザインの一部分やエレメントに異なる制約を指定できます。別のソースから設定された場合や別の HDL オブジェクトに設定された場合などに、ツールでどの制約が適用されるかを理解するには、「制約の優先順序」を参照してください。

制約の優先順序

一般的に、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されます。たとえば、信号 (またはインスタンス) とそれを含むデザインユニットの両方に制約を設定した場合、信号 (またはインスタンス) の制約が優先されます。同様に、信号 (またはインスタンスやエンティティ、モジュールなど) に設定された制約は XST コマンドラインや ISE® Design Suite で指定した制約よりも優先されます。

制約が同じオブジェクトに別の方法で設定されている場合は、次の順序がその優先度になります。

1. XST Constraint File (XCF)
2. Hardware Description Language (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

ISE Design Suite の合成オプション

このセクションでは、ISE® Design Suite の合成オプションについて説明します。

- ・ ISE Design Suite での XST オプションの設定
- ・ その他の XST コマンドライン オプションの設定
- ・ デザイン目標とストラテジ

ISE Design Suite での XST 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] パネルで [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のみが表示されます。
 - b. [Advanced]: 使用可能なオプションがすべて表示されます。
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 XST コマンド ライン 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. [Process Properties] ダイアログ ボックスを開きます。
2. [Synthesis Options] をクリックします。
3. [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。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。第 2 章「XST プロジェクトの作成および合成」の「XST のコメント」を参照してください。

デザイン目標とストラテジ

ISE® Design Suite (XST も含む) には、複数の目標やストラテジがあらかじめ含まれており、特定の最適化目標に合わせたオプションが既に定義されています。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 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。属性は使用する前に次の構文で宣言する必要があります。

```
attribute AttributeName : Type ;
```

VHDL

```
attribute RLOC : string ;
```

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できます。

- ・ 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。
- ・ VHDL 属性は宣言後、次のように指定します。

```
attribute AttributeName of ObjectList : ObjectType is AttributeValue ;
```

VHDL 属性の例

```
attribute RLOC of ul23 : label is "R11C1.S0" ;  
attribute bufg of my_signal : signal is "sr";
```

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。

- ・ **entity**
- ・ **architecture**
- ・ **component**
- ・ **label**
- ・ **signal**
- ・ **variable**
- ・ **type**

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。

Verilog-2001 の属性

このセクションでは、Verilog-2001 属性について説明します。

- ・ [Verilog-2001 属性の概要](#)
- ・ [Verilog-2001 の構文](#)
- ・ [Verilog-2001 の制限](#)
- ・ [Verilog のメタ コメント](#)

Verilog-2001 属性の概要

XST では Verilog-2001 属性文がサポートされています。属性は、合成ツールなどのアプリケーションに特定の情報を渡すために使用します。Verilog-2001 の属性は、モジュール宣言およびインスタンス化内で、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 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
- ・ 次のような特別な合成制約を設定する場合
 - [フル ケース \(FULL_CASE\)](#)
 - [パラレル ケース \(PARALLEL_CASE\)](#)

Verilog-2001 の構文

Verilog-2001 の属性は、(*) で囲む必要があり、次の構文で記述します。

```
(* attribute_name = attribute_value *)
```

この場合、それぞれ次を表します。

- ・ 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。このため、(*** attribute_name ***) は (*** attribute_name = "1" ***) と同じです。

attribute は、参照する信号、モジュール、またはインスタンスの宣言の直前に記述する必要があります。これには、別の行で次のように記述します。

```
(* ram_extract = "yes" *)
reg [WIDTH-1:0] myRAM [SIZE-1:0];
```

属性は、次のように同じ行に宣言として記述することもできます。

```
(* ram_extract = "yes" *) reg [WIDTH-1:0] myRAM [SIZE-1:0];
```

複数の属性をカンマで区切ったリストは、次のように記述できます。これらの属性は、同じ Verilog オブジェクトに適用されます。

```
(* attribute_name1 = attribute_value1, attribute_name2 = attribute_value2 *)
```

次のように指定することもできます。

```
(* 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 属性のコード例

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。

次のコード例は、1 つまたは複数のプロパティをモジュール、ポート、内部信号などにそれぞれ適用し、Verilog で属性を指定するさまざまな方法を示しています。また、**full_case** および **parallel_case** も属性構文を使用して case 文に適用されています。

```
//
// Verilog 2001 attribute examples
//
// Download: ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
// File: Verilog_Language_Support/attributes/vlgattrib2001_1.v
//
```

```
(* mux_extract = "no" *)
module vlgattrib2001_1 (clk, we1, we2, sel, re1, re2, waddr, raddr, di, do);

    (* max_fanout = "100", buffer_type = "none" *) input  clk;
    input      [1:0] sel;
    input      we1, we2;
    input      re1, re2;
    input      [7:0] waddr;
    input      [7:0] raddr;
    input      [15:0] di;
    output reg [15:0] do;

    (* mux_extract = "yes",
       use_clock_enable = "no" *)
    reg re;

    (*
       ram_extract = "yes",
       ram_style = "block"
    *)
    reg [15:0] RAM [255:0];

    (* keep = "true" *) wire    we;

    assign we = we1 | we2;

    always @ (posedge clk)
    begin
        (* full_case *) (* parallel_case *)
        case (sel)
            2'b00 : re <= re1 & re2;
            2'b01 : re <= re1 | re2;
            2'b10 : re <= re1;
            2'b11 : re <= re2;
        endcase
    end

    always @ (posedge clk)
    begin
        if (we)
            RAM[waddr] <= di;
        if (re)
            do <= RAM[raddr];
        end
    end

endmodule
```

Verilog-2001 の制限

次の Verilog-2001 属性は、サポートされていません。

- ・ 信号宣言
- ・ 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 コードで指定することもできます。Verilog-2001 属性構文が推奨されます。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
```

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"
```

次の制約には、異なる構文を使用します。

- ・ フル ケース (FULL_CASE)
- ・ パラレル ケース (PARALLEL_CASE)
- ・ 変換なし (TRANSLATE_OFF) と変換あり (TRANSLATE_ON)

詳細は、第 4 章「Verilog のサポート」の「Verilog 2001 の属性とメタ コメント」を参照してください。

XST Constraint File (XCF)

このセクションでは、XST Constraint File (XCF) について説明します。

- ・ XCF の概要
- ・ ネイティブ UCF 制約とそれ以外の UCF 構文
- ・ 構文の制限
- ・ XCF ファイルからのみ適用可能なタイミング制約

XCF の概要

HDL ソースコードで XST 制約を指定する以外にも、XST Constraint File (XCF) で指定する方法があります。XCF ファイルの拡張子は .xcf です。

ISE® Design Suite で XCF ファイルを指定するには

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。
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。
7. コマンドラインで XCF を指定するには、run コマンドで **(-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、第 2 章「XST プロジェクトの作成と合成」の「XST のコマンド」を参照してください。

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。また、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。

制約は 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で XST の run コマンドから定義し、例外を XCF ファイルで指定します。XCF ファイルで指定した制約は、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;
```

コード例

```
MODEL top mux_extract = false;
MODEL my_design max_fanout = 256;
```

上記の例では、デザインにエンティティ my_design を複数回インスタンス化すると、max_fanout=256 制約が my_design の各インスタンスに適用されます。

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。

構文は次のとおりです。

```
BEGIN MODEL entityname

INST instancename constraintname = constraintvalue ;
NET signalname constraintname = constraintvalue ;

END;
```

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は次の 2 タイプの UCF (ユーザー制約ファイル) 制約に分類できます。

- ・ [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 User Constraints File (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。次のようなネイティブ UCF 制約には、ワイルドカードおよび階層名を含めたネイティブ UCF 構文を使用します。

- ・ [周期 \(PERIOD\)](#)
- ・ [オフセット \(OFFSET\)](#)
- ・ [From-To \(FROM-TO\)](#)
- ・ [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 \(TIMEGRP\)](#)
- ・ [タイミング無視 \(TIG\)](#)

これらの制約は **BEGIN MODEL...** **END** 文内には使用しないでください。使用すると、XST でエラーが発生します。

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。

- ・ 次のような純粋な XST 制約
 - [FSM 自動抽出 \(FSM_EXTRACT\)](#)
 - [RAM スタイル \(RAM_STYLE\)](#)
- ・ 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 \(KEEP\)](#)

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、この区切り文字を使用します。XST で挿入される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。

構文の制限

XST Constraint File (XCF) 構文には、次の制限があります。

- ・ MODEL 文のネストはサポートされません。
- ・ **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。階層インスタンス名または信号名はサポートされません。
- ・ タイミング制約以外の制約では、インスタンス名や信号名のワイルドカードはサポートされません。
- ・ すべてのネイティブ User Constraints File (UCF) 制約がサポートされているわけではありません。詳細は、『[制約ガイド](#)』を参照してください。

XCF ファイルからのみ適用可能なタイミング制約

次の XST タイミング制約は、XST Constraint File (XCF) 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 (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 (TIMEGRP)
- ・ タイミング無視 (TIG)
- ・ タイミング スペック (**TIMESPEC**) (参照 : 『[制約ガイド](#)』)
- ・ タイミング スペック識別子 (**TSidentifier**) (参照 : 『[制約ガイド](#)』)

これらのタイミング制約はインプリメンテーション ツールに伝搬されるだけでなく、XST でも認識され、合成最適化に影響を与えます。これらの制約を配置配線 (PAR) に渡すには、[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

メモ：『XST ユーザー ガイド (Virtex-6 および Spartan-6 用)』は、Virtex®-6 および Spartan®-6 デバイス専用のマニュアルです。その他のデバイスを使用して XST を実行する場合は、『XST ユーザー ガイド』を参照してください。

この章では、XST の一般制約について次のセクションに分けて説明します。

- ・ I/O バッファの追加 (-iobuf)
- ・ ボックス タイプ (BOX_TYPE)
- ・ バスの区切り文字指定 (-bus_delimiter)
- ・ 大文字/小文字の指定(-case)
- ・ Case 文のインプリメンテーション形式 (-vlgcase)
- ・ Verilog マクロ (-define)
- ・ 複製接尾語の設定 (-duplication_suffix)
- ・ フル ケース (FULL_CASE)
- ・ RTL 回路図の生成 (-rtlview)
- ・ ジェネリック (-generics)
- ・ 階層区切り文字の指定 (-hierarchy_separator)
- ・ I/O 規格 (IOSTANDARD)
- ・ キープ (KEEP)
- ・ 階層の維持 (KEEP_HIERARCHY)
- ・ ライブラリの検索順 (-lso)
- ・ ロケーション (LOC)
- ・ ネットリスト階層 (-netlist_hierarchy)
- ・ 最適化エフォート レベル (OPT_LEVEL)
- ・ 最適化方法の指定 (OPT_MODE)
- ・ パラレル ケース (PARALLEL_CASE)
- ・ 相対ロケーション (RLOC)
- ・ 保存 (S / SAVE)
- ・ 合成制約ファイルの指定 (-uc)
- ・ 変換なし (TRANSLATE_OFF) と変換あり (TRANSLATE_ON)
- ・ 合成制約ファイルの無視 (-iuc)
- ・ Verilog の 'include ディレクトリの指定 (-vlgincdir)
- ・ HDL ライブラリ マップ ファイル (-xsthdpini)
- ・ 作業ディレクトリの指定 (-xsthdpdir)

I/O バッファの追加 (-iobuf)

-iobuf 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。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 バッファは各 I/O に手動でもインスタンスエートできます。その場合、I/O バッファがインスタンスエートされていない I/O にのみ I/O バッファが追加されます。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。このオプションは、後でインスタンスエートするデザインの一部を合成する際に便利です。

この制約に使用できる値は、次のとおりです。

- ・ **yes** (デフォルト)
- ・ **no**

yes を選択すると、IBUF および IOBUF プリミティブが生成されます。IBUF/OBUF プリミティブは、最上位レベル モジュールの I/O ポートに接続されます。大型デザインの後でインスタンス化される内部モジュールを合成するよう XST が呼び出される場合は、このオプションを no に設定する必要があります。デザインに I/O バッファを追加した場合は、この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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-iobuf {yes|no|true|false|soft}
```

デフォルトは、yes です。

ISE Design Suite からの設定

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] → [Add I/O Buffers]

ボックス タイプ (BOX_TYPE)

ボックス タイプ (BOX_TYPE) は、合成制約です。

この制約に使用できる値は、次のとおりです。

- **primitive**
- **black_box**
- **user_black_box**

これらの値により、XST でモジュールのビヘイビアが合成されないようにできます。

black_box は primitive と同じですが、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 を指定した場合、ログ ファイルにブラック ボックスのインスタンスがレポートされますが、primitive を指定した場合にはレポートされません。

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。これは Verilog および XCF のためにインプリメントされた制約で、VHDL で BOX_TYPE 制約をコンポーネントに適用した場合と同様の機能になります。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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

次をインスタンス化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;
```

バスの区切り文字指定 (-bus_delimiter)

信号ベクタをネットリストに書き込む際に使用するバス区切り文字の形式を指定します。指定できるバス区切り文字は次のとおりです。

- ・ <> (デフォルト)
- ・ []
- ・ {}
- ・ ()

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能エレメント

XST のコマンドラインで使います。

適用ルール

ありません。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-bus_delimiter {<>|[ ]|{}|() }
```

デフォルトは <> です。

ISE Design Suite からの設定

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 → [Synthesis Options] → [Bus Delimiter]

大文字/小文字の指定(-case)

-case を使用すると、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、名前に大文字を使用するか、小文字を使用するか、ソースの文字を保持するかを指定できます。ソースの文字は、Verilog または VHDL 合成フローで保持できます。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能エレメント

XST のコマンドラインで使います。

適用ルール

ありません。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-case {upper|lower|maintain}
```

デフォルトは [Maintain] です。

ISE Design Suite からの設定

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 → [Synthesis Options] → [Case]

Case 文のインプリメンテーション形式 (-vlgcase)

-vlgcase (Case 文のインプリメンテーション形式) は、Verilog デザインでのみ使用できます。

この制約を指定すると、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。次ののいずれかを選択します。

- ・ **full**
- ・ **parallel**
- ・ **full-parallel**

次の規則が適用されます。

- ・ オプションを指定しない場合
case 文のビヘイビアを記述どおりにインプリメントします。
- ・ **full**
case 文は完全と判断され、ラッチは作成されません。
- ・ **parallel**
case 文の分岐は同時に実行されないと判断され、プライオリティ エンコーダは使用されません。
- ・ **full-parallel**
case 文完全で分岐は同時に実行されないと判断され、ラッチおよびプライオリティ エンコーダは使用されません。

詳細は、次のセクションを参照してください。

- ・ [フル ケース \(FULL_CASE\)](#)
- ・ [パラレル ケース \(PARALLEL_CASE\)](#)
- ・ 詳細は、[第 7 章「HDL コーディング手法」](#)の「[マルチプレクサ](#)」を参照してください。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-vlgcase {full|parallel|full-parallel}
```

デフォルトでは、値は設定されていません。

ISE Design Suite からの設定

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 → [HDL Options] → [Case Implementation Style]

次ののいずれかを選択します。

- ・ **full**
- ・ **parallel**
- ・ **full-parallel**

デフォルトでは、値は設定されていません。

Verilog マクロ (–define)

これは、Verilog デザインにのみ使用できます。使用すると、Verilog マクロを定義または再定義でき、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。これは、IP コアの生成やフロー テストのようなプロセスで便利な機能です。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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-define {name[=value] name[=value] -}
```

この場合、それぞれ次を表します。

- ・ *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 {}
```

- ・ 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 からの設定

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 → [Synthesis Options] → [Verilog Macros]

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"
```

複製接尾語の設定 (-duplication_suffix)

-duplication_suffix を使用すると、フリップフロップが複製されたときの XST の名前の付け方を設定できます。XST でフリップフロップが複製される際、元のフリップフロップ名に _n (n は整数) が付きます。

たとえば、元のフリップフロップ名が my_ff の場合、同じフリップフロップが 3 度複製されると、名前はそれぞれ次のようになります。

- ・ **my_ff_1**
- ・ **my_ff_2**
- ・ **my_ff_3**

このオプションを使用すると、デフォルト名の終わりにテキスト スtring を追加できます。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。

たとえば、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`

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。

たとえば、`_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`

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-duplication_suffix string%dstring
```

デフォルトは `%d` です。

ISE Design Suite からの設定

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 → [Synthesis Options] → [Other]

フル ケース (FULL_CASE)

これは、Verilog デザインにのみ使用できます。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。この制約を使用すると、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、[第 7 章「HDL コーディング手法」](#)の「[マルチプレクサ](#)」を参照してください。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

Verilog の構文例

Verilog 2001 の構文は次のとおりです。

(* full_case *)

コマンドにはターゲット参照が含まれないため、セレクタ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'b001x: res = data3;
  4'b0001: 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、セレクタのすぐ後ろにメタ コメントを記述します。

```
casex select // synthesis full_case
  4'b1xxx: res = data1;
  4'b01xx: res = data2;
  4'b001x: res = data3;
  4'b0001: res = data4;
endcase
```

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。構文は次のとおりです。

-vlgcase {full|parallel|full-parallel}

ISE Design Suite からの設定

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 → [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 (-rtlview)

デザインの RTL 構造を表すネットリストファイルを生成します。生成されたネットリストは、RTL Viewer および Technology Viewer で表示できます。RTL 回路図の生成は、次のいずれかの値に設定できます。

- ・ **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。RTL 表示を含むファイルの拡張子は .ngr です。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-rtlview {yes|no|only}
```

デフォルトは no です。

ISE Design Suite からの設定

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 → [Synthesis Options] → [Generate RTL Schematic]

デフォルトは、yes です。

ジェネリック (-generics)

使用すると、最上位レベルのデザイン ブロックで定義したジェネリック (VHDL) またはパラメータ (Verilog) の値を定義し直すことができます。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。これは、IP コアの生成やフロー テストのようなプロセスで便利な機能です。

定義された値が HDL ソースコードで定義されたデータ型と合わない場合、XST でコマンド ラインの定義が無視されることを示す警告メッセージが表示されます。

データ型の違いが XST で認識されなかった場合は、警告メッセージは表示されず、HDL ファイルで定義したデータ型で値が処理されます。指定する値は HDL ソースコードで定義されたデータ型と一致するようにしてください。定義されたジェネリックまたはパラメータの名前がデザインに存在しない場合は、何のメッセージも表示されず、その定義は無視されます。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

XST コマンド ラインの構文例

```
xst run -generics {name=value name=value ...}
```

この場合、それぞれ次を表します。

- ・ *name* は、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の名前を示します。
- ・ *value* は、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の値を示します。

デフォルトでは何も定義されていません。

```
-generics {}
```

- ・ {} 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**

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。

XST コマンド ラインの構文フォーマット

タイプ	ジェネリック値の構文例
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

接頭語とその値の間にはスペースを入れないでください。

```
-generics {company="Xilinx" width=5 init_vector=b100101}
```

このコマンドは、次を設定しています。

- ・ company を Xilinx に設定
- ・ width を 5 に設定
- ・ init_vector を b100101 に設定

階層区切り文字の指定 (-hierarchy_separator)

-hierarchy_separator 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。

サポートされる文字は次の 2 つです。

- ・ _ (アンダースコア)
- ・ / (スラッシュ)

新規プロジェクトでのデフォルトは / (スラッシュ) です。

たとえば、デザインにインスタンス INST1 というサブブロックがあり、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。階層がフラット化される場合に、このオプションがアンダースコアに設定されていると、TMP_NET は INST1_TMP_NET という名前になります。このオプションがスラッシュに設定されている場合は、ネット名は INST1/TMP_NET となります。

階層の区切り文字としてスラッシュを使用した方が階層名を識別しやすいので、デバッグの際に便利です。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-hierarchy_separator {_|/}
```

新規プロジェクトでのデフォルトは / (スラッシュ) です。

ISE Design Suite からの設定

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 → [Synthesis Options] → [Hierarchy Separator]

デフォルトはスラッシュ (/) です。

I/O 規格 (IOSTANDARD)

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。I/O 規格は VHDL 属性、Verilog 属性、XCF 制約を使用して信号やインスタンスに個別に適用できます。グローバルには適用されません。

詳細は、『[制約ガイド](#)』の「IOSTANDARD」を参照してください。

キープ (KEEP)

KEEP 制約は、高度なマップ制約です。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。これは、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 を使用すると、これが回避できます。

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、構造が保持されるわけではありません。たとえば、デザインに 2 ビットのマルチプレクサ セレクタがあり、これに KEEP 制約を設定した場合、この信号は最終ネットリストに保持されますが、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、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 制約だけでなく、[列挙型エンコード手法 \(ENUM_ENCODING\)](#) も使用する必要があります。

この制約には、次の値が使用できます。

- **true**
- **soft**
- **false**

soft 値を使用すると、合成中に該当する信号を保持できますが、KEEP 制約はインプリメンテーションに渡されず、信号が最適化で削除されます。

KEEP 制約は VHDL 属性、Verilog 属性、XCF 制約を使用して信号に適用できます。

詳細は、『[制約ガイド](#)』の「KEEP」を参照してください。

階層の維持 (KEEP_HIERARCHY)

KEEP_HIERARCHY は、合成およびインプリメンテーション制約です。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階層が保持され、この階層を含むシミュレーション ネットリストが生成されます。

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。この制約を true (有効) に設定すると、階層ネットリストを作成でき、デザインのエンティティとモジュールの階層およびインターフェイスを保持できます。

この制約は、HDL 合成で推論されたマクロではなく、HDL デザインで指定された階層ブロック (VHDL のエンティティ、Verilog のモジュール) に適用されます。

KEEP_HIERARCHY の値

この制約に使用できる値は、次のとおりです。

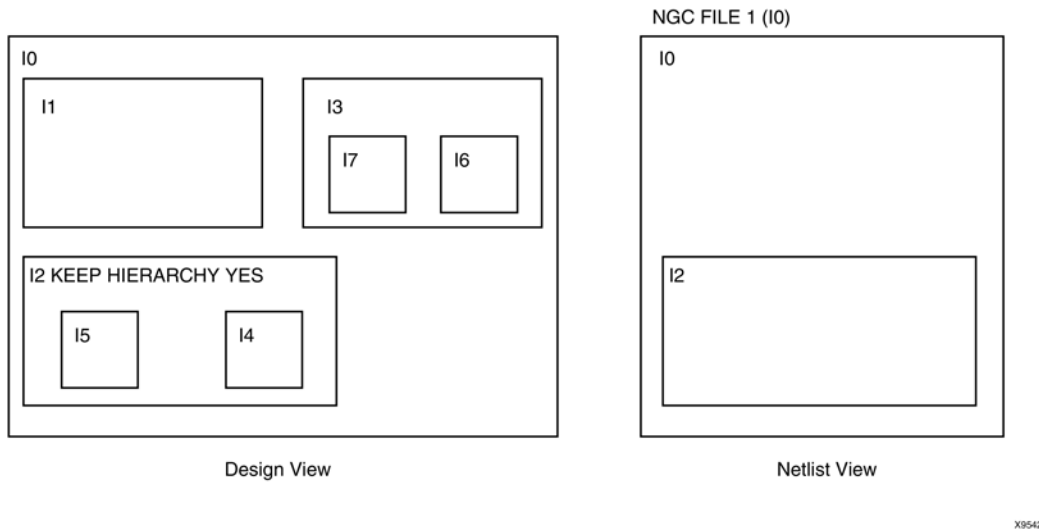
- ・ **true**
HDL プロジェクトで記述されたデザイン階層を保持します。この値を合成に適用した場合、インプリメンテーションにも適用されます。
- ・ **false** (デフォルト)
階層ブロックが 最上位モジュールにマージされます。
- ・ **soft**
合成ではデザイン階層が保持されますが、インプリメンテーションには制約は適用されません。

階層の保持

一般的に、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。また、コラプスや因数分解などの最適化のプロセスはロジック全体にグローバルに適用されるため、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 (積項およびデバイス マクロセルの少量化、周波数の向上など)。

階層維持の図

たとえば、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 の階層は維持されたまま最後のネットリストに含まれますが、I2 の下にある 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。



アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

回路図

- ・ エンティティまたはモジュール シンボルに設定します。
- ・ 属性名：**KEEP_HIERARCHY**
- ・ 属性値：**YES, NO**

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|true|false|soft}";
```

デフォルトは no です。

Verilog の構文例

```
(* keep_hierarchy = "{yes|no|true|false|soft}" *)
```

XCF の構文例

```
MODEL "entity_name" keep_hierarchy={yes|no|true|false|soft};
```

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。構文は次のとおりです。

```
-keep_hierarchy {yes|no|soft}
```

デフォルトは no です。

詳細は、第 2 章「XST プロジェクトの作成および合成」の「コマンドライン モードからの XST の実行」を参照してください。

ISE Design Suite からの設定

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 → [Synthesis Options] → [Keep Hierarchy]

ライブラリの検索順 (-lso)

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。

- ISE® Design Suite の [Process Properties] ダイアログ ボックスの [Synthesis Options] ページにある [Library Search Order]
- lso コマンドライン オプション

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-lso file_name.lso
```

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。

詳細は、第 6 章「混合言語のサポート」の「ライブラリ検索順 (LSO) ファイル」を参照してください。

ISE Design Suite からの設定

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 → [Synthesis Options] → [Library Search Order] を選択します。

詳細は、第 6 章「混合言語のサポート」の「ライブラリ検索順 (LSO) ファイル」を参照してください。

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。詳細は、『制約ガイド』の「LOC」を参照してください。

ネットリスト階層 (-netlist_hierarchy)

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します。これを使用すると、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。

値は、次のいずれかになります。

- ・ **as_optimized**

XST では、**階層の維持 (KEEP_HIERARCHY)** 制約が考慮され、NGC ネットリストを最適化された形式で生成します。このモードの場合、階層ブロック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で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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
- netlist_hierarchy {as_optimized|rebuilt}
```

デフォルトは as_optimized です。

最適化エフォート レベル (OPT_LEVEL)

合成最適化のエフォートレベルを指定します。

この制約に使用できる値は、次のとおりです。

- ・ **1** (標準の最適化)
特に階層デザインで、処理が高速になります。ザイリンクスでは、ほとんどのデザインにこのレベル 1 (標準) を推奨しています。デフォルトは 1 です。
- ・ **2** (高度な最適化)
さらに多くの最適化手法をイネーブルにするには、2 を使用します。これらの手法を使用すると、合成のランタイムがかなり増加してしまうことがあり、必ずしも結果が改善されるわけではありません。これらの最適化は特定のデザインには有効ですが、全く改善がない場合や、結果が悪化することすらあります。このため、ほとんどのデザインで最適化エフォートレベルは 1 をお勧めします。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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 "{1|2}";
```

Verilog の構文例

```
(* opt_level = "{1|2}" *)
```

XCF の構文例

```
MODEL "entity_name" opt_level={1|2};
```

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。構文は次のとおりです。

```
-opt_level {1|2}
```

デフォルトは 1 です。

ISE Design Suite からの設定

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 → [Synthesis Options] → [Optimization Effort]

最適化方法の指定 (OPT_MODE)

OPT_MODE を使用すると、合成の最適化方法を指定できます。

次の値のいずれかを選択できます。

- **speed**
この値を設定すると、ロジックレベル数の低減が優先され、周波数が向上します。これがデフォルトです。
- **area**
この値を設定すると、デザインのインプリメンテーションに使用されるロジックの総数を低減することが優先されるため、デザイン フィットが向上します。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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 コマンドラインの構文例

run コマンドでグローバルに設定します。構文は次のとおりです。

-opt_mode {area|speed}

デフォルトは、speed です。

ISE Design Suite からの設定

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 → [Synthesis Options] → [Optimization Goal]

デフォルトは、speed です。

パラレル ケース (PARALLEL_CASE)

これは、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、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。詳細は、[第 7 章「HDL コーディング手法」](#)の「[マルチプレクサ](#)」を参照してください。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

Verilog の構文例

構文は次のとおりです。

(* 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'blxxx: res = data1;
  4'bxlxx: res = data2;
  4'bxxlx: res = data3;
  4'bxxx1: 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'blxxx: res = data1;
  4'bxlxx: res = data2;
  4'bxxlx: res = data3;
  4'bxxx1: res = data4;
endcase
```

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。構文は次のとおりです。

```
-vlgcase {full|parallel|full-parallel}
```

RLOC

RLOC 制約は、基本的なマップおよび配置の制約です。RLOC 制約は、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。詳細は、『制約ガイド』の「RLOC」を参照してください。

保存 (S / SAVE)

この制約は、高度なマップ制約です。通常は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たり、LUT のような ELEMENT が最適化で削除されることがあります。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護する必要がある場合は、SAVE 制約を使用すると、このような最適化が回避できます。また、ネットやブロックの複製、レジスタの自動調整といった最適化手法も SAVE 制約によってオフにできます。

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 ELEMENT 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 ELEMENT に接続されたネットも保存されます。

SAVE 制約が LUT のようなブロックに適用されると、その LUT は直接接続された信号すべてと共に保存されます。

適用できる ELEMENT は次のとおりです。

- ・ 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 ELEMENT すべてと共に最終ネットリストに保存されます。この結果、これらの ELEMENT 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
S (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 は直接接続された信号すべてと共に保存されます。

詳細は、『制約ガイド』を参照してください。

合成制約ファイルの指定 (-uc)

合成で使用する XST 合成制約ファイル (XCF) を指定します。この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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-uc filename
```

ISE Design Suite からの設定

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 → [Synthesis Options] → [Synthesis Constraints File]

変換なし (TRANSLATE_OFF) と変換あり (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: 合成を再開する点を指定

TRANSLATE_OFF および TRANSLATE_ON は Synplicity および Synopsys の制約でもあり、XST では Verilog でサポートされています。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**

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。ただし、Verilog 構文は通常のメタ コメント構文とは異なり、次のようになります。

```
// synthesis translate_off
...code not synthesized...
// synthesis translate_on
```

合成制約ファイルの無視 (-iuc)

この制約を使用すると、合成制約ファイル (-uc) で指定した制約ファイルが合成中に無視されます。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-iuc {yes|no}
```

デフォルトは no です。

ISE Design Suite からの設定

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 → [Synthesis Options] → [Synthesis Constraints File]

Verilog の ‘include ディレクトリの指定 (-vlgincdir)

-vlgincdir を使用すると、‘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を検索します。

- ・ 現在のディレクトリ
- ・ inc ディレクトリ
- ・ 現在のファイルの相対ディレクトリ

メモ： -vlgincdir は、‘include と共に使用してください。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

XST コマンド ラインの構文例

run コマンドで -vlgincdir オプションを使用してグローバルに設定します。指定できる値はディレクトリ名です。詳細は、[第 2 章「XST プロジェクトの作成および合成」の「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。

run コマンドでグローバルに設定します。構文は次のとおりです。

```
-vlgincdir {directory_path [directory_path]}
```

デフォルト値はありません。

ISE Design Suite からの設定

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 → [Synthesis Options] → [Verilog Include Directories]

指定できる値はディレクトリ名です。

デフォルト値はありません。

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 ライブラリ マップ ファイル (-xsthdpini)

-xsthdpini を使用すると、ライブラリのマップに使用するファイルを選択できます。

ライブラリ マップ ファイルには、次の 2 つの関連パラメータがあります。

- ・ **XSTHDPINI**
- ・ **XSTHDPDIR**

ライブラリ マップ ファイルには、次が含まれます。

- ・ 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。

- ・ ザイリンクスのソフトウェアをインストールするとインストールされるあらかじめ定義されたファイル
- ・ ユーザーがプロジェクトに合わせて定義できるユーザー ファイル

あらかじめ定義されているデフォルトの INI ファイルの名前は `xhdp.ini` で、`%XILINX%\vhdl\xst` にあります。このファイルには、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。このファイルは変更できませんが、ユーザー ライブラリ マップを作成する際に同じ構文を使用できます。このファイルの内容は次のとおりです。

```
-- 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
```

この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 ファイルの場所は、次のいずれかの方法で指定できます。

- ・ ISE® Design Suite の次のプロパティから VHDL 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
```

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
set -xsthdpini file_name
```

このコマンドで指定できるファイルは 1 つのみです。

ISE Design Suite からの設定

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 → [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。

作業ディレクトリの指定 (-xsthdpdir)

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指定します。このプロパティは、次のいずれかの方法で設定します。

- ・ ISE® Design Suite の次のプロパティから設定：
[Process Properties] ダイアログ ボックス → [Synthesis Options] → [VHDL Work Directory]、または
- ・ 次のコマンドを使用して、コマンド ラインから設定します。

```
set -xsthdpdir directory
```

具体例

次の状況であると仮定します。

- ・ 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。

この場合、この 3 人のユーザーは次を設定する必要があります。

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
```

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
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。

デフォルト値はありません。

ISE Design Suite からの設定

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 → [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

メモ: 『XST ユーザー ガイド (Virtex-6 および Spartan-6 用)』は、Virtex®-6 および Spartan®-6 デバイス専用のマニュアルです。その他のデバイスを使用して XST を実行する場合は、『XST ユーザー ガイド』を参照してください。

この章では、XST の HDL 制約について次のセクションに分けて説明します。

- ・ FSM 自動抽出 (FSM_EXTRACT)
- ・ 列挙型エンコード手法 (ENUM_ENCODING)
- ・ 等価レジスタの削除 (EQUIVALENT_REGISTER_REMOVAL)
- ・ FSM エンコード方法の指定 (FSM_ENCODING)
- ・ MUX の抽出 (MUX_EXTRACT)
- ・ MUX の最小サイズ (MUX_MIN_SIZE)
- ・ リソース共有 (RESOURCE_SHARING)
- ・ セーフ リカバリ ステート (SAFE_RECOVERY_STATE)
- ・ セーフ インプリメンテーション (SAFE_IMPLEMENTATION)

FSM 自動抽出 (FSM_EXTRACT)

FSM_EXTRACT を使用すると、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または無効にできます。FSM_ENCODING 制約および FSM_FFTYPE 制約の値を設定するには、この制約を使用する必要があります。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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、モジュール、または信号に適用されます。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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 コマンドラインの構文例

run コマンドでグローバルに設定します。構文は次のとおりです。

```
-fsm_extract {yes|no}*
```

デフォルトは、yes です。

ISE Design Suite からの設定

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 → [HDL Options] → [FSM Encoding Algorithm]

オプションは次のとおりです。

- ・ **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。**-fsm_encoding** の詳細については、「[FSM エンコード方法の指定 \(FSM_ENCODING\)](#)」を参照してください。

列挙型エンコード手法 (ENUM_ENCODING)

ENUM_ENCODING を使用すると、VHDL 列挙型に適用するエンコード手法を選択できます。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。この制約は、列挙型に 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タイプを使用して Finite State Machine (FSM) を記述する場合、ENUM_ENCODING でエンコード方法を指定する必要があります。指定したエンコードが XST で使用されるようにするには、ステートレジスタの [FSM エンコード 方法 \(FSM_ENCODING\)](#) を user に設定する必要があります。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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 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 (EQUIVALENT_REGISTER_REMOVAL)

EQUIVALENT_REGISTER_REMOVAL を使用すると、RTL レベルで記述された等価レジスタの削除を有効または無効にできます。デフォルトでは、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、デザインがターゲット デバイスにフィットする可能性が上がります。

この制約に使用できる値は、次のとおりです。

- ・ **yes** (デフォルト)
フリップフロップの最適化が有効です。
- ・ **no**
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アルゴリズム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。
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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 コマンドラインの構文例

run コマンドでグローバルに設定します。構文は次のとおりです。

```
-equivalent_register_removal {yes | no}
```

デフォルトは、yes です。

ISE Design Suite からの設定

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] → [Equivalent Register Removal]

FSM エンコード方法の指定 (FSM_ENCODING)

FSM_ENCODING 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。

この制約に使用できる値は、次のとおりです。

- ・ `auto`
- ・ `one-hot`
- ・ `compact`
- ・ `sequential`
- ・ `gray`
- ・ `johnson`
- ・ `speed1`
- ・ `user`

デフォルトは `auto` です。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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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、モジュール、または信号に適用されます。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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}";
```

デフォルトは、`auto` です。

Verilog の構文例

Place immediately before the module or signal declaration.

```
(* fsm_encoding = "{auto|one-hot |compact|sequential|gray|johnson|speed1|user}" *)
```

デフォルトは、`auto` です。

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 ラインの構文例

run コマンドでグローバルに設定します。構文は次のとおりです。

```
-fsm_encoding {auto|one-hot|compact|sequential|gray|johnson|speed1|user}
```

デフォルトは、auto です。

ISE Design Suite からの設定

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 → [HDL Options] → [FSM Encoding Algorithm]

指定可能なタイプは、次のとおりです。

- ・ [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 (FSM_EXTRACT)」を参照してください。

MUX の抽出 (MUX_EXTRACT)

MUX_EXTRACT を使用すると、マルチプレクサ マクロの推論を有効または無効にできます。マルチプレクサはよく使用される機能で、合成後の結果は XST での特定マルチプレクサ マクロの推論が適切かどうかの決定によってかなり異なります。マルチプレクサのマクロ推論は、さまざまなデザインで最適な結果になるように設定されていますが、MUX_EXTRACT を使用して、デザインの指定部分に対するツールのデフォルト決定を上書きでき、XST によるデフォルトのソリューションを改善することも可能です。

設定できる値は、次のいずれかです。

- ・ **yes** または **true** (XCF のみ)
- ・ **no** または **false** (XCF のみ)
- ・ **force**

デフォルトでは、マルチプレクサの推論は yes に設定されています。XST では、各マルチプレクサの記述に対し、内部決定ルールに従ってマクロが作成されるか、または残りの組み合わせロジックと共に最適化されます。force および no に設定すると、このデフォルトのルールが無視され、force の場合は XST で強制的にマクロが作成され、no の場合はマクロが作成されないようにできます。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能エレメント

VHDL エンティティ、Verilog モジュール、または信号にのみ選択して適用します。コマンドライン オプションや ISE® Design Suite のオプションでグローバルには制御できません。

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

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

VHDL の構文例

次のように宣言します。

```
attribute mux_extract: string;
```

次のように指定します。

```
attribute mux_extract of {signal_name | entity_name}: {entity|signal} is  
"{yes|no|force}";
```

デフォルトは、yes です。

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。

```
(* mux_extract = "{yes|no|force}" *)
```

デフォルトは、yes です。

XCF の構文例 1

エンティティ またはモジュールに次のように設定します。

```
MODEL "entity_name" mux_extract={yes|no|true|false|force};
```

XCF の構文例 2

信号に次のように設定します。

```
BEGIN MODEL "entity_name"  
NET "signal_name" mux_extract={yes|no|true|false|force};  
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。構文は次のとおりです。

```
-mux_extract {yes|no|force}
```

デフォルトは、yes です。

MUX の最小サイズ (MUX_MIN_SIZE)

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**) 制約を使用すると、XST で推論されるマルチプレクサ マクロの最小サイズを制御できます。

「サイズ」は、マルチプレクサを通るデータ入力の数です。たとえば、2:1 マルチプレクサの場合、サイズ (マルチプレクサを通る入力の数) は 2 で、16:1 マルチプレクサの場合は 16 になります。セレクト入力にはカウントしません。

この数は、選択したデータの幅とは無関係です。1 ビット幅の 8:1 マルチプレクサと 16 ビット幅の 8:1 マルチプレクサの両方とも、サイズは 8 になります。

この制約には 1 よりも大きな整数値を指定できます。デフォルト値は 2 です。

デフォルトでは、XST は 2:1 マルチプレクサ マクロを推論します。2:1 マルチプレクサの明示的推論には、デザインによって最後の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、この制約の使用はお勧めしません。

デバイス使用率に問題があり、その大きな原因となるデザイン部分に推論される 2:1 マルチプレクサが多くある場合、この制約を使用することでデバイス使用率が改善されます。この場合、2:1 マルチプレクサの推論をグローバルに、または結果に特に影響するブロックに対して、ディスエーブルにすることをお勧めします。2:1 マルチプレクサの推論をディスエーブルにするには、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。このような状況で MUX_MIN_SIZE を使用する場合には、十分な注意が必要です。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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";
```

デフォルトは 2 です。

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" *)
```

デフォルトは 2 です。

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。構文は次のとおりです。

```
-mux_min_size integer
```

メモ : 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 (RESOURCE_SHARING)

RESOURCE_SHARING 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できます。

設定可能な値は、次のいずれかです。

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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 resynthesize 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 ラインの構文例

run コマンドでグローバルに設定します。構文は次のとおりです。

```
-resource_sharing {yes|no}
```

デフォルトは、yes です。

ISE Design Suite からの設定

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 → [HDL Options] → [Resource Sharing]

セーフ リカバリ ステート (SAFE_RECOVERY_STATE)

SAFE_RECOVERY_STATE 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 で指定したステート

詳細は、「[セーフ インプリメンテーション \(SAFE_IMPLEMENTATION\)](#)」を参照してください。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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

設定した信号に適用されます。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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;
```

セーフ インプリメンテーション (SAFE_IMPLEMENTATION)

SAFE_IMPLEMENTATION 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。このモードでは、FSM が無効なステートになった場合に有効なステート (リカバリ ステート) に戻すロジックが追加されます。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。

[セーフ リカバリ ステート \(SAFE_RECOVERY_STATE\)](#) 制約を適用すると、手動でリカバリ ステートを定義できます。

セーフ インプリメンテーションは、次のいずれかの方法で指定できます。

- 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。
- 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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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 コマンドラインの構文例

run コマンドでグローバルに設定します。構文は次のとおりです。

```
-safe_implementation {yes|no}
```

デフォルトは no です。

ISE Design Suite からの設定

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 → [HDL Options] → [Safe Implementation]

FPGA 制約（タイミング制約以外）

メモ：『XST ユーザー ガイド (Virtex-6 および Spartan-6 用)』は、Virtex®-6 および Spartan®-6 デバイス専用のマニュアルです。その他のデバイスを使用して XST を実行する場合は、『XST ユーザー ガイド』を参照してください。

この章では、XST の FPGA 制約について次のセクションに分けて説明します。

- ・ 非同期から同期への変換 (ASYNC_TO_SYNC)
- ・ 自動 BRAM パッキング (AUTO_BRAM_PACKING)
- ・ BRAM 使用率 (BRAM_UTILIZATION_RATIO)
- ・ バッファ タイプ (BUFFER_TYPE)
- ・ BUFGCE の抽出 (BUFGCE)
- ・ コアの検索ディレクトリ (-sd)
- ・ DSP 使用率 (DSP_UTILIZATION_RATIO)
- ・ FSM スタイル (FSM_STYLE)
- ・ 電力削減 (POWER)
- ・ コアの読み込み (READ_CORES)
- ・ 再合成 (RESYNTHESIZE)
- ・ インクリメンタル合成 (INCREMENTAL_SYNTHESIS)
- ・ LUT の結合
- ・ BRAM へのロジックのマッピング (BRAM_MAP)
- ・ 最大ファンアウト数 (MAX_FANOUT)
- ・ 最初のフリップフロップ ステージの移動 (MOVE_FIRST_STAGE)
- ・ 最後のフリップフロップ ステージの移動 (MOVE_LAST_STAGE)
- ・ 乗算器スタイル (MULT_STYLE)
- ・ グローバル クロック バッファ数 (-bufg)
- ・ インスタンス化されたプリミティブの最適化 (OPTIMIZE_PRIMITIVES)
- ・ I/O レジスタの IOB 内へのパック (IOB)
- ・ RAM の抽出 (RAM_EXTRACT)
- ・ RAM スタイル (RAM_STYLE)
- ・ 制御セットの削減 (REDUCE_CONTROL_SETS)
- ・ レジスタの自動調整 (REGISTER_BALANCING)
- ・ レジスタの複製 (REGISTER_DUPLICATION)

- ・ ROM の抽出 (ROM_EXTRACT)
- ・ ROM スタイル (ROM_STYLE)
- ・ シフトレジスタの抽出 (SHREG_EXTRACT)
- ・ スライス パッキング (-slice_packing)
- ・ ロー スキュー ラインの使用 (USELOWSKEWLINES)
- ・ スライス (LUT-FF ペア) の使用率
- ・ SLICE_UTILIZATION_RATIO_MAXMARGIN (スライス (LUT-FF ペア) 使用率の許容範囲)
- ・ 単一 LUT へのエンティティのマッピング (LUT_MAP)
- ・ キャリーチェーンの使用 (USE_CARRY_CHAIN)
- ・ トライステートからロジックへの変換 (TRISTATE2LOGIC)
- ・ クロック イネーブルの使用 (USE_CLOCK_ENABLE)
- ・ 同期セットの使用 (USE_SYNC_SET)
- ・ 同期リセットの使用 (USE_SYNC_RESET)
- ・ DSP ブロックの使用 (USE_DSP48)

多くの場合、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 (ASYNC_TO_SYNC)

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。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。

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。厳密な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。自動で非同期から同期への変換を実行させると、HDL ソースコードのシーケンシャル 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。デザインに含まれる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

注意！ 非同期から同期へ変換するこの制約を使用すると、次のような影響が出ます。

- ・ 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が、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、またはそれらが同期ソースから派生していない場合、合成後のソリューションの機能は同じになります。
- ・ この変換によりデザイン目標を達成したら、予測通りの回路動作にするために、HDL 記述を変更して同期の set および reset 信号にする必要があります。HDL 記述を変更すると、設計検証もやりやすくなります。
- ・ 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推奨します。
- ・ コーディング方法を変更できるのであれば、同期 set および reset 信号を HDL ソース コードで記述することをお勧めします。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-async_to_sync {yes|no}
```

デフォルトは no です。

ISE Design Suite からの設定

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 → [HDL Options] → [Asynchronous to Synchronous]

自動 BRAM パッキング (AUTO_BRAM_PACKING)

この制約を使用すると、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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-auto_bram_packing {yes|no}
```

デフォルトは no です。

ISE Design Suite からの設定

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 → [Automatic BRAM Packing]

BRAM 使用率 (BRAM_UTILIZATION_RATIO)

合成中に XST で処理されるブロック RAM 数を制限できます。デザインに含まれるブロック RAM にはブロック RAM の推論プロセスからだけでなく、インスタンス化とブロック RAM マップ最適化からのものもあります。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、第 8 章「FPGA の最適化」の「ブロック RAM へのロジックのマップ」を参照してください。

インスタンス化されたブロック 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。ただし、インスタンス化されたブロック 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 でインスタンス化が修正されず、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、警告メッセージが表示され、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。この方法は、特定デザインで潜在的に推論されるブロック RAM の数を確認するために使用できます。

デザインに含まれるブロック RAM 数がターゲット 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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-block RAM_utilization_ratio <integer>[%][#]
```

この場合、それぞれ次を表します。

<integer> の範囲は -1 ~ 100

および

範囲は、% が使用されるか、% と # のどちらも削除されます。

デフォルトは 100 です。

XST コマンド ラインの構文例 1

```
-bram_utilization_ratio 50
```

これは次を意味します。

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%
```

これは次を意味します。

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#
```

これは次を意味します。

50 ブロック RAM

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。

XST で推論されるブロック RAM の数を確認する場合などは、このブロック RAM のリソース自動リソース管理オプションをオフにすることもできます。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 からの設定

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 の個数で指定する形式はサポートされていません。

バッファ タイプ (BUFFER_TYPE)

この制約は、CLOCK_BUFFER に替わる制約です。CLOCK_BUFFER 制約は今後サポートされなくなるので、この制約を使用することをお勧めします。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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能エレメント

信号に適用されます。

適用ルール

設定した信号に適用されます。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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  
"{bufp11|ibufg|bufg|bufgp|bufh|bufr|bufio|bufio2fb|bufio2|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 =  
"{bufp11|ibufg|bufg|bufgp|bufh|bufr|bufio|bufio2fb|bufio2|ibuf|obuf|buf|none}" *)
```

XCF の構文例

```
BEGIN MODEL "entity_name "
```

```
NET "signal_name "
```

```
buffer_type={bufp11|ibufg|bufg|bufgp|bufh|bufr|bufio|bufio2fb|bufio2|ibuf|obuf|buf|none};
```

```
END;
```

BUFGCE の抽出 (BUFGCE)

BUFGMUX プリミティブを推論し、BUFGMUX の機能をインプリメントします。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。

使用できる値は、次のいずれかです。

- ・ **yes**
- ・ **no**

BUFGCE の抽出は、HDL コードで設定できます。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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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

設定した信号に適用されます。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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

```
BEGIN MODEL "entity_name"
```

```
NET "primary_clock_signal" bufgce={yes|no|true|false};
```

```
END;
```

コアの検索ディレクトリ (-sd)

-sd を使用すると、デフォルトのディレクトリ以外に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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-sd {directory_path [directory_path]}
```

指定できる値はディレクトリ名です。詳細は、[第 2 章「XST プロジェクトの作成および合成」の「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。

デフォルト値はありません。

ISE Design Suite からの設定

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 → [Synthesis Options] → [Cores Search Directory]

DSP 使用率 (DSP_UTILIZATION_RATIO)

DSP_UTILIZATION_RATIO を使用すると、合成最適化で使用可能な DSP スライスの絶対数またはパーセントを指定できます。デフォルトでは、ターゲット デバイスの 100% に設定されています。

デザインに含まれる DSP スライスには DSP の推論からだけでなく、インスタンス化からのものもあります。インスタンス化された DSP スライスは使用可能な DSP リソースの第一候補として認識され、DSP が推論されると、残りの DSP リソースに配置されます。インスタンス化された DSP の数が使用可能なリソース数を上回ってしまう場合、XST でインスタンス化が修正されず、それらはブロック DSP スライスとしてはインプリメントされません。[DSP ブロックの使用 \(USE_DSP48\)](#) を使用して特定のマクロを DSP スライス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、DSP スライ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 DSP スライスの数がターゲット デバイスの使用可能な DSP リソース数を上回る場合は、XST で警告メッセージが表示され、チップ上の使用可能な DSP リソースのみが合成に使用されます。

XST で推論される DSP の数を確認する場合などは、-1 (または負の値) を設定して、この DSP のリソース自動リソース管理オプションをオフにすることもできます。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

-dsp_utilization_ratio number[%|#]

この場合、それぞれ次を表します。

<integer> の範囲は -1 ~ 100 です。

これは、次の条件での範囲です。

範囲は、% が使用されるか、% と # のどちらも削除されます。

合計スライスの割合を指定するには % を、スライスの絶対値を指定する場合は、# を使用します。

デフォルトは % です。

- ターゲット デバイスの DSP ブロック数の 50% に設定する場合は、次のように入力します。

```
-dsp_utilization_ratio 50
```

- ターゲット デバイスの DSP ブロック数の 50% に設定する場合は、次のように入力します。

```
-dsp_utilization_ratio 50%
```

- DSP ブロック数を 50 個に設定する場合は、次のように入力します。

```
-dsp_utilization_ratio 50#
```

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。

ISE Design Suite からの設定

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 → [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。

FSM スタイル (FSM_STYLE)

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。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。

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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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、モジュール、または信号に適用されます。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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}";
```

デフォルトは lut です。

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 からの設定

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 → [Synthesis Options] → [FSM Style]

電力削減 (POWER)

この制約を使用すると、消費電力をできる限り抑えることができます。最低限の電力でファンクションをインプリメントされるように、マクロ プロセスが実行されます。このオプションは、AREA モードとも SPEED モードとも併用できますが、最終的に全体のエリアやスピードに悪影響を与えることもあります。

現在のリリースでは、DSP48 とブロック RAM にしか XST で電力最適化を設定できません。

XST では、次の 2 つのブロック RAM 最適化方法がサポートされます。

- ・ 方法 1: エリアやスピードにそれほど影響はありません。これは、電力削減制約が使用される場合のデフォルトです。
- ・ 方法 2: 電力を削減しますが、スピードに影響が出ます。

どちらの方法も `RAM_STYLE` 制約を使用します。方法 1 の場合は `block_power1` を、方法 2 の場合は `block_poewr2` を使用します。

デザインの改善方法を示す HDL Advisor メッセージが表示されることがあります。たとえば、XST でブロック RAM に `READ_FIRST` モードが設定されているのが検出されると、`WRITE_FIRST` または `NO_CHANGE` モードへの変更を勧めるメッセージが表示されることがあります。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能エレメント

次に適用されます。

- ・ 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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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}";
```

デフォルトは no です。

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。

```
(* power = "{yes|no}" *)
```

デフォルトは no です。

XCF の構文例

```
MODEL "entity_name" power = {yes|no|true|false};
```

デフォルトは false です。

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。構文は次のとおりです。

```
-power {yes|no}
```

デフォルトは no です。

ISE Design Suite からの設定

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 → [Synthesis Options] → [Power Reduction]

コアの読み込み (READ_CORES)

READ_CORES 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、Electronic Data Interchange Format (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が認識されるので、XST でそのコアの周囲のロジックを最適化されやすくなります。

必要な結果を出すために READ_CORES をオフにする必要のあることもあります。たとえば、PCI™ コアの最適化はほかのコアとは異なる方法で最適化する必要があります。この制約を使用すると、コア別にコアを読み込むかどうかを指定できます。

詳細は、第 8 章「FPGA の最適化」の「コアの処理」を参照してください。

次のいずれかを指定します。

- ・ **no (false)**
コアはプロセスされません。
- ・ **yes (true)**
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。
- ・ **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、XST コマンドラインを使用した場合にのみ使用できます。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能エレメント

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、両方の制約を適用できるオブジェクト セットは同じである必要があります。

次に適用されます。

- ・ 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)

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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}";
```

デフォルトは、yes です。

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。

```
(* read_cores = "{yes|no|optimize}" *)
```

デフォルトは、yes です。

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 ラインの構文例

```
-read_cores {yes|no|optimize}
```

ISE Design Suite からの設定

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 → [Synthesis Options] → [Read Cores]

ISE Design Suite からは、optimize オプションは指定できません。

再合成 (RESYNTHESIZE)

インクリメンタル合成 (INCREMENTAL_SYNTHESIS) 制約で作成したグループの再合成をするかどうかを指定します。

設定できる値は次のとおりです。

この制約に使用できる値は、次のとおりです。

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

グローバルには設定できません。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能エレメント

デザイン エレメントのみに適用されます。

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

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

VHDL の構文例

次のように宣言します。

```
attribute resynthesize: string;
```

次のように指定します。

```
attribute resynthesize 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。

```
(* resynthesize = "{yes|no}" *)
```

XCF の構文例

```
MODEL "entity_name" resynthesize={yes|no};
```

インクリメンタル合成 (INCREMENTAL_SYNTHESIS)

この制約はインクリメンタル合成フローで使用し、デザインの分割方法を指定します。VHDL エンティティまたは Verilog モジュールに適用でき、各エンティティ/モジュールに対して NGC ファイルが 1 つずつ生成されます。パーティションの詳細は、ISE® Design Suite ヘルプを参照してください。

この制約は [Process Properties] ダイアログ ボックスからは設定できず、次からのみ設定できます。

- ・ VHDL 属性
- ・ Verilog メタ コメント
- ・ XST Constraint File (XCF)

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

VHDL の構文例

次のように宣言します。

```
attribute incremental_synthesis: string;
```

次のように指定します。

```
attribute incremental_synthesis 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。

```
(* incremental_synthesis = "{yes|no}" *)
```

XCF の構文例

```
MODEL "entity_name" incremental_synthesis={yes|no};
```

LUT の結合 (LC)

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により、デザイン速度が削減できることもあります。

この制約には、次の 3 つの値が設定できます。

- ・ **auto**
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。Virtex®-6 および Spartan®-6 デバイス ファミリでは auto がデフォルトです。
- ・ **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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-lc {auto|area|off}
```

ISE Design Suite からの設定

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 → [Synthesis Options] → [LUT Combining]

BRAM へのロジックのマッピング (BRAM_MAP)

階層ブロック全体を、Virtex® 以降のデバイスに搭載されているブロック RAM リソースにマッピングするよう指定します。設定できる値は次のいずれかです。

- ・ **yes**
- ・ **no** (デフォルト)

この制約は、グローバルにもローカルにも適用できます。

詳細は、第 8 章「FPGA の最適化」の「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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能エレメント

BRAM に適用されます。

適用ルール

RAM にマッピングするロジック (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ピングされません。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 が推論されない場合、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ピングされていることを確認してください。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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ネーション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;
```

最大ファンアウト数 (MAX_FANOUT)

MAX_FANOUT 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。値は整数にします。デフォルト値は 10 万です。MAX_FANOUT は、グローバルにもローカルにも設定できます。

ファンアウトが大きいと配線で問題を生じることがあるため、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。ゲートを複製できない場合は、バッファが挿入されます。これらのバッファは、NGC ファイルで **キープ (KEEP)** を設定すると、インプリメンテーション レベルの最適化で削除されないようになります。

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。

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 のビヘイビアが異なります。

- MAX_FANOUT の値を ISE® Design Suite またはコマンドラインを使用して設定するか、特定の階層ブロックに適用した場合、XST ではこの値が基準として解釈されます。
- 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かったことがあります。

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。MAX_FANOUT を ISE Design Suite で設定している場合は、XST がタイミングを向上しようとしてネットを複製する場合があります。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。

MAX_FANOUT には、**reduce** という値も設定できます。この値を設定しても XST には直接影響しませんが、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。

MAX_FANOUT の **reduce** 値はネットにのみ指定できます。グローバルには指定できません。

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。これは、合成後のネットリストに保存され、**MAX_FANOUT=reduce** プロパティが指定されていることを意味します。

さらに多くのグローバルな MAX_FANOUT 制約を integer 値で定義した場合は (コマンドラインを使用して設定するか、該当する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)、次のようになります。

- 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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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" *)
```

Or

```
(* 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 ラインの構文例

```
-max_fanout integer
```

ISE Design Suite からの設定

ISE Design Suite で次のようにグローバルに定義します。

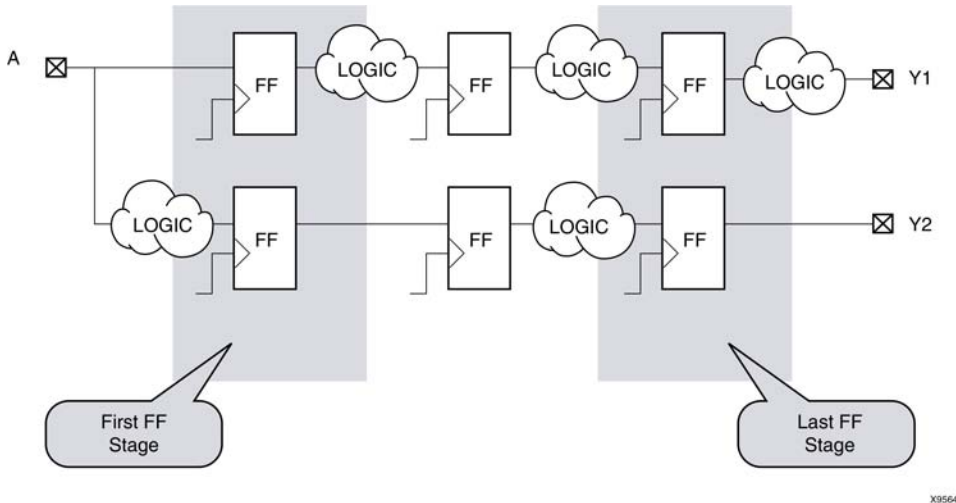
[Process Properties] ダイアログ ボックス → [Xilinx-Specific Options] → [Max Fanout]

最初のフリップフロップ ステージの移動 (MOVE_FIRST_STAGE)

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。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。

- ・ フリップフロップ (図の FF) は、そのパスがプライマリ入力から接続されている場合は最初のフリップフロップ 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。

最初のフリップフロップ ステージの移動



レジスタ バランス (自動調整) の間、フリップフロップはそれぞれ次の方向に移動します。

- ・ 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。これを防ぐには、次の場合に OFFSET_IN_BEFORE および OFFSET_IN_AFTER を使用します。

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、次のいずれかの値に設定できます。**yes** および **no**。

- ・ MOVE_FIRST_STAGE を no に設定すると、最初の段階にあるフリップフロップは移動しません。
- ・ MOVE_LAST_STAGE を no に設定すると、最終の段階にあるフリップフロップは移動しません。

複数の制約を付けると、レジスタのバランス プロセスに影響があります。詳細は、「[レジスタの自動調整 \(REGISTER_BALANCING\)](#)」を参照してください。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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

上の図を参照してください。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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 ラインの構文例

run コマンドでグローバルに設定します。構文は次のとおりです。

```
-move_first_stage {yes|no}
```

デフォルトは、yes です。

ISE Design Suite からの設定

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] → [Move First Flip-Flop Stage]

最後のフリップフロップ ステージの移動 (MOVE_LAST_STAGE)

MOVE_LAST_STAGE を使用すると、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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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

「最初のフリップフロップ ステージの移動 (MOVE_FIRST_STAGE)」を参照してください。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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 コマンドラインの構文例

run コマンドでグローバルに設定します。構文は次のとおりです。

```
-move_last_stage {yes|no}
```

デフォルトは、yes です。

ISE Design Suite からの設定

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] → [Move Last Flip-Flop Stage]

乗算器スタイル (MULT_STYLE)

MULT_STYLE を使用すると、乗算器マクロのインプリメント方法を指定できます。

この制約に使用できる値は、次のとおりです。

- ・ **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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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、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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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}";
```

デフォルトは、auto です。

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}" *)
```

デフォルトは、auto です。

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 バッファ数 (-bufg)

<imk 74>XST で作成する BUFG の最大数を整数で指定します。値は整数にします。デフォルト値はデバイスによって異なり、そのデバイスの **BUFG** 。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-bufg integer
```

値は整数にします。次表に示すように、デフォルト値はデバイスによって異なります。アーキテクチャ別のデフォルト値は、次の表を参照してください。ターゲット 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。

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

ISE Design Suite からの設定

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] → [Number of Clock Buffers]

インスタンス化されたプリミティブの最適化 (OPTIMIZE_PRIMITIVES)

デフォルトでは、HDL コードに含まれるインスタンス化されたプリミティブは最適化されません。OPTIMIZE_PRIMITIVES を使用すると、このデフォルト設定を解除でき、HDL にインスタンス化されたザイリンクス ライブラリ プリミティブが XST で最適化できるようになります。

インスタンス化したプリミティブの最適化には、次のような制限があります。

- ・ 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。
- ・ 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、インスタンス化したプリミティブの最適化が設定されていても、最適化 (変更) されません。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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
- ・ 属性値
 - **yes**
 - **no** (デフォルト)

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 ラインの構文例

run コマンドでグローバルに設定します。構文は次のとおりです。

```
-optimize_primitives {yes|no}
```

デフォルトは no です。

ISE Design Suite からの設定

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] → [Optimize Instantiated Primitives]

I/O レジスタの IOB 内へのパック (IOB)

IOB 制約を使用すると、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 内に配置できます。

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**

タイミング制約でカバーされないと判断された IOB にレジスタが含まれるので、タイミングの最適化が考慮されません。たとえば、**PERIOD** 制約を指定した場合、XST では PERIOD 制約でカバーされないレジスタが IOB に含まれます。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。

詳細は、『[制約ガイド](#)』の「IOB」を参照してください。

RAM の抽出 (RAM_EXTRACT)

RAM_EXTRACT 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。

設定できる値は次のいずれかです。

この制約に使用できる値は、次のとおりです。

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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、モジュール、または信号に適用されます。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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 コマンドラインの構文例

run コマンドでグローバルに設定します。構文は次のとおりです。

```
-ram_extract {yes|no}
```

デフォルトは、yes です。

ISE Design Suite からの設定

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 → [HDL Options] → [RAM Extraction]

RAM スタイル (RAM_STYLE)

RAM_STYLE を使用すると、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。

RAM スタイル制約に設定できる値は次のいずれかです。

- ・ **auto** (デフォルト)
- ・ **block**
- ・ **distributed**
- ・ **pipe_distributed**
- ・ **block_power1**
- ・ **block_power2**

推論された各 RAM 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。

電力重視でブロック RAM の最適化をするには、block_power1 および block_power2 を使用します。詳細は、「[電力削減 \(POWER\)](#)」を参照してください。

インプリメンテーションにブロック RAM か分散 RAM ソースを使用するように手動で設定できます。

pipe_distributed、block_power1、block_power2 は、HDL か XCF 制約のいずれかで指定できます。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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、モジュール、または信号に適用されます。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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}";
```

デフォルトは、auto です。

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}" *)
```

デフォルトは、auto です。

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 ラインの構文例

run コマンドでグローバルに設定します。構文は次のとおりです。

```
-ram_style {auto|block|distributed}
```

デフォルトは、auto です。

コマンド ラインからは、pipe_distributed には設定できません。

ISE Design Suite からの設定

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 → [HDL Options] → [RAM Style]

制御セットの削減 (REDUCE_CONTROL_SETS)

REDUCE_CONTROL_SETS を使用すると、制御セットの数を削減でき、デザイン エリアの削減につながります。制御セット数を削減すると、map のパッキング プロセスが改善されるので、LUT 数が増加した場合でも、使用されるスライス数が減少されます。

この制約は同期制御信号 (同期セット/リセットおよびクロック イネーブル) にのみ適用され、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。

制約の値には、次のいずれかを使用してください。

- **auto** (デフォルト)
XST により自動的に最適化され、デザインに含まれる制御セットが削減されます。
- **no**
制御セットの最適化は実行されません。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-reduce_control_sets {auto|no}
```

ISE Design Suite からの設定

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 → [Synthesis Options] → [Reduce Control Sets]

レジスタの自動調整 (REGISTER_BALANCING)

REGISTER_BALANCING を使用すると、レジスタ自動調整 (リタイミング) を有効または無効にできます。レジスタ自動調整では、クロック周波数を向上するため、ロジックに対してフリップフロップおよびラッチの位置を移動します。

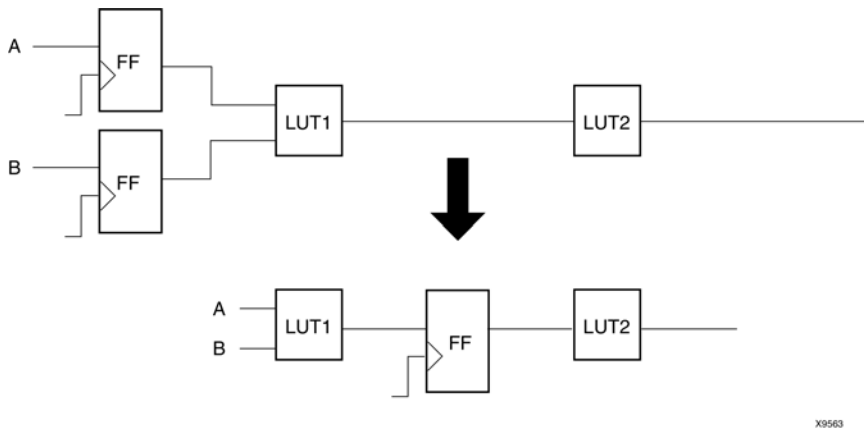
レジスタ自動調整には、次の 2 種類があります。

- ・ 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。

順方向のレジスタ自動調整



X9563

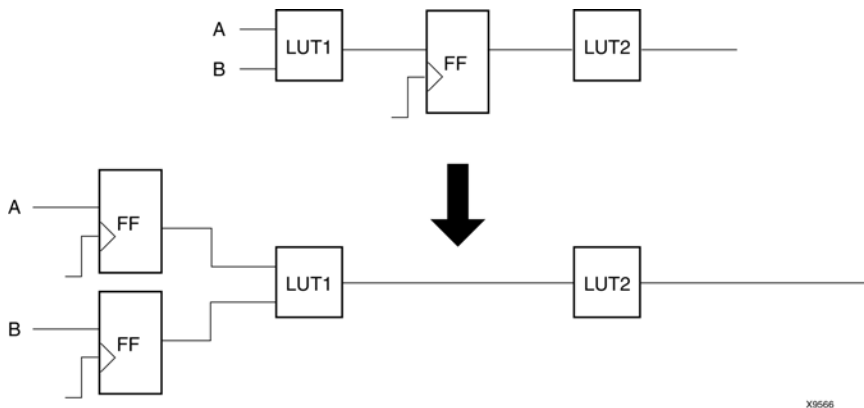
複数のフリップフロップが 1 つのフリップフロップに置き換わる際には、次のように LUT 名に基づいた名前が選択されます。

*LutName_***FRB***Id*

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。

逆方向のレジスタ自動調整



X9566

この結果、デザインのフリップフロップ数が増減します。

新しいフリップフロップには、次のように元のフリップフロップ名の後に接尾辞が付きます。

*OriginalFFName_***BRB***Id*

レジスタ自動調整の値

この制約に使用できる値は、次のとおりです。

- ・ **yes**
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** (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。
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

レジスタ自動調整に影響するその他の制約

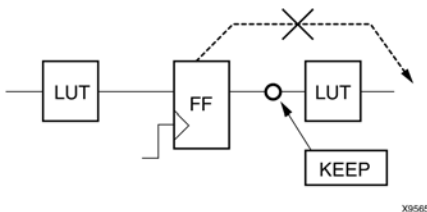
次の制約もレジスタ自動調整に影響を与えます。

- ・ [最初のフリップフロップ ステージの移動 \(MOVE_FIRST_STAGE\)](#)
- ・ [最後のフリップフロップ ステージの移動 \(MOVE_LAST_STAGE\)](#)

また、次の制約もレジスタ自動調整に影響を与えます。

- ・ [階層の維持 \(KEEP_HIERARCHY\)](#)
 - 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。

出力フリップフロップに適用した場合



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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能エレメント

次に適用できます。

- ・ 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
- ・ フリップフロップ記述 (RTL) に対応する信号
- ・ フリップフロップ インスタンス
- ・ プライマリ クロック信号

この場合、レジスタ自動調整はフリップフロップがこのクロックと同期した場合にのみ実行されます。

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

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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}" *) (
```

デフォルトは no です。

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。構文は次のとおりです。

```
-register_balancing {yes|no|forward|backward}
```

デフォルトは no です。

ISE Design Suite からの設定

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 (REGISTER_DUPLICATION)

REGISTER_DUPLICATION 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。

設定できる値は次のいずれかです。

この制約に使用できる値は、次のとおりです。

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

デフォルトは、yes です。この場合、タイミング最適化およびファンアウト制御の段階でレジスタの複製がされます。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

VHDL の構文例

次のように宣言します。

```
attribute register_duplication: string;
```

次のように指定します。

```
attribute register_duplication 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。

```
(* 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 ラインの構文例

run コマンドでグローバルに設定します。構文は次のとおりです。

```
-register_duplication {yes|no}
```

デフォルトは、yes です。

ISE Design Suite からの設定

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 Duplication]

ROM の抽出 (ROM_EXTRACT)

ROM_EXTRACT を使用すると、ROM マクロの推論を有効または無効にできます。

設定できる値は次のいずれかです。

この制約に使用できる値は、次のとおりです。

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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 ラインの構文例

run コマンドでグローバルに設定します。構文は次のとおりです。

```
-rom_extract {yes|no}
```

デフォルトは、yes です。

ISE Design Suite からの設定

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 → [HDL Options] → [ROM Extraction]

ROM スタイル (ROM_STYLE)

ROM_STYLE を使用すると、推論された ROM マクロのインプリメント方法を指定できます。

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。

この制約に使用できる値は、次のとおりです。

- ・ **auto** (デフォルト)
- ・ **block**
- ・ **distributed**

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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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、モジュール、または信号に適用されます。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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}";
```

デフォルトは、auto です。

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}" *)
```

デフォルトは、auto です。

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。

run コマンドでグローバルに設定します。構文は次のとおりです。

```
-rom_style {auto|block|distributed}
```

デフォルトは、auto です。

ISE Design Suite からの設定

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 → [HDL Options] → [ROM Style]

シフト レジスタの抽出 (SHREG_EXTRACT)

SHREG_EXTRACT を使用すると、シフトレジスタ マクロの推論を有効または無効にできます。

設定できる値は、次のいずれかです。

この制約に使用できる値は、次のとおりです。

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

このオプションを使用すると、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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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

設定したデザイン エレメントまたは信号に適用されます。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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 コマンドラインの構文例

run コマンドでグローバルに設定します。構文は次のとおりです。

```
-shreg_extract {yes | no}
```

デフォルトは、yes です。

ISE Design Suite からの設定

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 → [HDL Options] → [Shift Register Extraction]

スライス パッキング (-slice_packing)

-slice_packing を使用すると、XST に含まれる内部パック機能を有効にできます。内部パックは、クリティカルな LUT 間の接続をスライスまたは CLB 内にパックしようとする機能です。これにより、CLB 内の LUT 間の高速フィードバック接続が使用されます。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-slice_packing {yes|no}
```

ISE Design Suite からの設定

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] → [Slice Packing]

ロー スキュー ラインの使用 (USELOWSKEWLINES)

USELOWSKEWLINES 制約は、基本的な配線制約です。合成の段階では、[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、すべてのネットに対してロー スキュー配線リソースが使用されるように指定します。詳細は、[『制約ガイド』](#)の「USELOWSKEWLINES」を参照してください。

Slice (LUT-FF Pairs) Utilization Ratio (スライス (LUT-FF ペア) 使用率)

SLICE_UTILIZATION_RATIO を使用すると、タイミング最適化におけるエリア サイズの上限を、LUT-FF ペアの合計の絶対値またはパーセントで指定します。

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、[第 8 章「FPGA の最適化」](#)の「[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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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

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

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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#";
```

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。

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#" *)
```

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。

XCF の構文例 1

```
MODEL "entity_name" slice_utilization_ratio=integer;
```

XCF の構文例 2

```
MODEL "entity_name" slice_utilization_ratio="integer%";
```

XCF の構文例 3

```
MODEL "entity_name" slice_utilization_ratio="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。

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。

% および # は XST Constraint File (XCF) 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。

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。構文は次のとおりです。

```
-slice_utilization_ratio integer  
-slice_utilization_ratio integer%
```

`-slice_utilization_ratio 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 -1 ~ 100 です。

ISE Design Suite からの設定

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 → [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 ではこの値を % 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。

SLICE_UTILIZATION_RATIO_MAXMARGIN (スライス (LUT-FF ペア) 使用率の許容範囲)

SLICE_UTILIZATION_RATIO_MAXMARGIN は、スライス (LUT-FF ペア) 使用率 (SLICE_UTILIZATION_RATIO) に関連する制約です。この制約では、スライス (LUT-FF ペア) 使用率 (SLICE_UTILIZATION_RATIO) の許容範囲を設定します。値は、パーセントのほか、LUT/FF ペアかスライスの絶対数で指定できます。

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、第 8 章「FPGA の最適化」の「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。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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

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

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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#";
```

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。

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。

```
(* 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。

XCF の構文例 1

```
MODEL "entity_name" slice_utilization_ratio_maxmargin=integer;
```

XCF の構文例 2

```
MODEL "entity_name" slice_utilization_ratio_maxmargin="integer%";
```

XCF の構文例 3

```
MODEL "entity_name" 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 LUT/FF 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 XST Constraint File (XCF) 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 ラインの構文例

run コマンドでグローバルに設定します。構文は次のとおりです。

```
-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 LUT/FF 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。

単一 LUT へのエンティティのマッピング (LUT_MAP)

1 つのブロックを 1 つの LUT にマッピングするように指定します。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。

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。

これには、LUT INIT ファンクションおよびその他の方法を使用できます。たとえば、1 つの LUT にマップするファンクションを HDL ソース コードの別ブロックで記述する方法があります。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。LUT の INIT 値は XST により自動的に算出され、最適化中この LUT が保持されます。

詳細は、第 8 章「FPGA の最適化」の「LUT へのロジックのマップ」を参照してください。

XST では、Synplicity の XC_MAP 制約が自動的に認識されます。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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化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};
```

キャリーチェーンの使用 (USE_CARRY_CHAIN)

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。USE_CARRY_CHAIN 制約を使用すると、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。この制約は、グローバルまたはローカルに設定できます。

設定できる値は次のいずれかです。

- ・ **yes** (デフォルト)
- ・ **no**

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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

設定した信号に適用されます。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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

- ・ 属性値
 - **yes**
 - **no**

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

Place immediately before the signal declaration.

```
(* 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 ラインの構文例

run コマンドでグローバルに設定します。構文は次のとおりです。

-use_carry_chain {yes|no}

デフォルトは、yes です。

トリステートからロジックへの変換 (TRISTATE2LOGIC)

デバイスによっては内部トリステートがサポートされないで、トリステートが自動的に等価ロジックに変換されます。トリステートから生成されるロジックは、周辺のロジックと組み合わせて最適化が可能なので、内部トリステートをロジックに置換すると、スピードを向上でき、場合によってはエリア最適化も向上できます。ただし、通常はトリステートをロジックに置換すると、エリアは増加します。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

- ・ ロジックに変換されるのは、内部トリステートのみです。出力パッドに接続された最上位モジュールのトリ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、内部トリステートはロジックに変換されません。
- ・ 次の場合、XST でトリステートがロジックに変換されません。
 - トリステートがブラックボックスに接続されている
 - トリステートがロジックの出力に接続され、そのブロックの階層が保護されている
 - トリステートが最上位レベルの出力に接続されている
 - トリステートが配置されたブロックまたはトリステートが接続された信号で TRISTATE2LOGIC が no に設定されている

この制約に使用できる値は、次のとおりです。

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能エレメント

次に適用されます。

- ・ 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)
- ・ 1 つの信号

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、信号、インスタンスに適用されます。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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 ラインの構文例

run コマンドでグローバルに設定します。構文は次のとおりです。

```
-tristate2logic {yes|no}
```

デフォルトは、yes です。

ISE Design Suite からの設定

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] → [Convert Tristates to Logic]

クロック イネーブルの使用 (USE_CLOCK_ENABLE)

USE_CLOCK_ENABLE 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。ASIC プロトタイプの場合は、通常クロック イネーブルを無効にします。

制約値を `no` または `false` 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 (CE) 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 CE 機能を付けることで、ロジック最適化が向上し、優れた結果品質 (QoR) を実現できることがあります。auto に設定すると、フリップフロップ入力の専用 CE 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 CE ロジックを使用した方がいいかが比較検討されます。フリップフロップがインスタンス化されると、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 イネーブルを削除します。

この制約に使用できる値は、次のとおりです。

- ・ `auto` (デフォルト)
- ・ `yes`
- ・ `no`
- ・ `true` (XCF のみ)
- ・ `false` (XCF のみ)

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能エレメント

次に適用されます。

- ・ 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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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 ラインの構文例

run コマンドでグローバルに設定します。構文は次のとおりです。

```
-use_clock_enable {auto|yes|no}
```

デフォルトは、auto です。

ISE Design Suite からの設定

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] → [Use Clock Enable]

同期セットの使用 (USE SYNC SET)

```

      . ASIC      ,      .      no      false
      ,
      ,      .      ,      ,
      ,      ,      (QoR)      .

```

auto に設定すると、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が比較検討されます。フリップフロップがインスタンス化されると、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、同期リセットを削除します。

設定できる値は次のいずれかです。

この制約に使用できる値は、次のとおりです。

- ・ **auto** (デフォルト)
- ・ **yes**
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能エレメント

次に適用されます。

- ・ 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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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 ラインの構文例

run コマンドでグローバルに設定します。構文は次のとおりです。

```
-use_sync_set {auto|yes|no}
```


デフォルトは、auto です。

ISE Design Suite からの設定

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] → [Use Synchronous Set]

同期リセットの使用 (USE_SYNC_RESET)

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します。ASIC プロトタイプをの場合は、同期セットを無効にできます。

制約値をno または false に設定すると、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機能を付けることで、ロジック最適化が向上し、優れた結果品質 (QoR) を実現できることがあります。

auto に設定すると、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が比較検討されます。フリップフロップがインスタンス化されると、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、同期リセットを削除します。

この制約に使用できる値は、次のとおりです。

- ・ **auto** (デフォルト)
- ・ **yes**
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能エレメント

次に適用されます。

- ・ 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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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 ラインの構文例

run コマンドでグローバルに設定します。構文は次のとおりです。

```
-use_sync_reset {auto|yes|no}
```

デフォルトは、auto です。

ISE Design Suite からの設定

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] → [Use Synchronous Reset]

DSP ブロックの使用 (USE_DSP48)

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します。

デフォルトは、auto です。

デフォルト値の auto を使用した場合、乗算、乗算加減算、乗累算などのマクロは DSP ブロ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などのマクロには、スライスにインプリメントされるものもあります。これらのマクロを DSP ブロックにインプリメントするには、この制約を yes または true に設定する必要があります。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。

MAC など DSP ブロックに配置できるマクロは、乗算器、アキュムレータ、レジスタなどの単純なマクロから構成されています。最適なパフォーマンスを得るため、XST はマクロ 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。マクロを特定の方法でインプリメントするには、**キープ (KEEP)** 制約を使用する必要があります。たとえば、DSP ブロックには 2 つの入力レジスタを使用した乗算器をインプリメントできますが、最初のレジスタ ステージを DSP48 の外に残したままにするには、出力に**キープ (KEEP)** 制約を設定する必要があります。

この制約に使用できる値は、次のとおりです。

- ・ **auto** (デフォルト)
- ・ **yes**
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

また、auto モードにすると、DSP 使用率 (DSP_UTILIZATION_RATIO) 制約を使用して合成で使用可能な DSP ブロック リソースの数が制御されます。デフォルトでは、使用可能な DSP ブロック リソースが可能な限り使用されます。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能エレメント

次に適用されます。

- ・ 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)
- ・ 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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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|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|yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" use_dsp48={auto|yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name "  
NET "signal_name" use_dsp48={auto|yes|no|true|false};  
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。構文は次のとおりです。

```
-use_dsp48 {auto|yes|no}
```

デフォルトは、auto です。

ISE Design Suite からの設定

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 → [HDL Options] → [Use DSP Block]

タイミング制約

メモ: 『XST ユーザー ガイド (Virtex-6 および Spartan-6 用)』は、Virtex®-6 および Spartan®-6 デバイス専用のマニュアルです。その他のデバイスを使用して XST を実行する場合は、『XST ユーザー ガイド』を参照してください。

この章では、XST のタイミング制約について次のセクションに分けて説明します。

- ・ タイミング制約の指定
- ・ クロス クロック解析 (-cross_clock_analysis)
- ・ タイミング制約の書き込み (-write_timing_constraints)
- ・ クロック信号 (CLOCK_SIGNAL)
- ・ グローバルな最適化目標 (-glob_opt)
- ・ XCF のタイミング制約のサポート
- ・ 周期 (PERIOD)
- ・ オフセット (OFFSET)
- ・ From-To (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 (TIMEGRP)
- ・ タイミング無視 (TIG)

タイミング制約の指定

このセクションでは、タイミング制約の適用について説明します。

- ・ タイミング制約の指定の概要
- ・ グローバル最適化オプションを使用したタイミング制約の指定
- ・ ユーザー制約ファイル (UCF) を使用したタイミング制約の指定
- ・ NGC ファイルへの制約の書き込み
- ・ タイミング制約の処理に影響のあるその他のオプション

タイミング制約の指定の概要

XST でサポートされるタイミング制約は、次の方法で指定できます。

- ・ グローバルな最適化目標 (-glob_opt)
- ・ [Process] → [Process Properties] → [Synthesis Options] → [Global Optimization Goal]
- ・ ユーザー制約ファイル (UCF)

グローバル最適化オプションを使用したタイミング制約の指定

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`

これらの制約は、デザイン全体にグローバルに適用されます。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。これらの制約は、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指定

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 (TIMEGRP)
- ・ 周期 (PERIOD)
- ・ タイミング無視 (TIG)
- ・ From-To (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。

NGC ファイルへの制約の書き込み

デフォルトでは、タイミング制約は 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。

- ・ [Process Properties] ダイアログ ボックスの [Synthesis Options] ページにある [Write Timing Constraints] をオン
- ・ `-write_timing_constraints` コマンドライン オプションを使用

タイミング制約の処理に影響のあるその他のオプション

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。

- ・ クロス クロック解析 (`-cross_clock_analysis`)
- ・ タイミング制約の書き込み (`-write_timing_constraints`)
- ・ クロック信号 (CLOCK_SIGNAL)

クロス クロック解析 (`-cross_clock_analysis`)

タイミングの最適化中に複数のクロックドメイン間を解析します。デフォルトでは `no` に設定されており、解析は実行されません。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-cross_clock_analysis {yes|no}
```

デフォルトは no です。

ISE Design Suite からの設定

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 ボックスの [Synthesis Options] ページにある [Cross Clock Analysis]

タイミング制約の書き込み (-write_timing_constraints)

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 の [Process Properties] ダイアログ ボックスの [Synthesis Options] ページにある [Write Timing Constraints] をオン
- ・ -write_timing_constraints コマンドライン オプションを使用

デフォルトでは、タイミング制約は NGC ファイルに書き込まれません。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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。構文は次のとおりです。

```
-write_timing_constraints {yes|no}
```

デフォルトは no です。

ISE Design Suite からの設定

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 → [Synthesis Options] → [Write Timing Constraints]

クロック信号 (CLOCK_SIGNAL)

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、クロック信号となる入力ピンまたは内部信号が認識されません。この制約を設定すると、クロック信号を定義できます。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

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。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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;
```


グローバルな最適化目標 (-glob_opt)

XST では、この制約に基づいて次のデザイン部分を最適化できます。

- ・ 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

この制約では、グローバルな最適化目標を選択できます。サポートされるタイミング制約の詳細は、ISE® Design Suite ヘルプの「パーティション」を参照してください。

この制約には、値を指定できません。最適なパフォーマンスを得るため、デザイン全体が最適化されます。

次の制約を適用できます。

- ・ **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。

これらの制約はデザイン全体に影響し、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、適用されます。

run コマンドで -glob_opt オプションを使用してグローバルに設定できます。構文は次のとおりです。

```
glob_opt {allclocknets|offset_in_before|offset_out_after  
|inpad_to_outpad|max_delay} -
```

この値は、ISE Design Suite の [Process Properties] ダイアログ ボックスの [Synthesis Options] ページにある [Global Optimization Goal] で設定できます。

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。

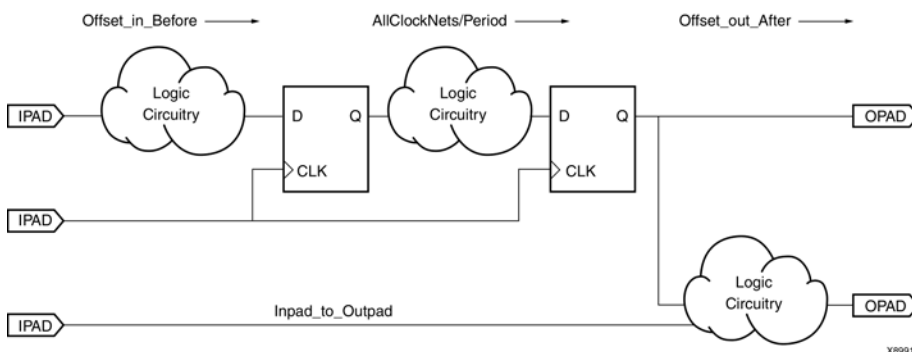
- ・ INPAD_TO_OUTPAD (入力パッドから出力パッド)

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



XCF のタイミング制約のサポート

XST Constraint File (XCF) ファイルでタイミング制約を指定する場合、階層の区切り記号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。詳細については、「[階層区切り文字](#)」を参照してください。

指定したタイミング制約またはその一部が XST でサポートされていない場合、警告メッセージが表示され、タイミング最適化段階でその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ています。

- ・ 周期 (PERIOD)
- ・ オフセット (OFFSET)
- ・ From-To (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 (TIMEGRP)
- ・ タイミング無視 (TIG)

周期 (PERIOD)

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。詳細は、『[制約ガイド](#)』の「PERIOD」を参照してください。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能ELEMENT

詳細は、『[制約ガイド](#)』の「PERIOD」を参照してください。

適用ルール

詳細は、『[制約ガイド](#)』の「PERIOD」を参照してください。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

XCF の構文例

```
NET netname PERIOD = value [{HIGH|LOW} value];
```

オフセット (OFFSET)

OFFSET は、基本的なタイミング制約です。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。この制約はパッド関連の信号のみに使用され、デザインの内部信号への信号到着時間は指定できません。

オフセット制約は次のために使用します。

- ・ 外部ネットからデータ入力とクロック入力 that 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 ピンからクロックが供給され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できます。

詳細は、『[制約ガイド](#)』の「OFFSET」を参照してください。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能エレメント

詳細は、『[制約ガイド](#)』の「OFFSET」を参照してください。

適用ルール

詳細は、『[制約ガイド](#)』の「OFFSET」を参照してください。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

XCF の構文例

```
OFFSET = {IN|OUT} offset_time [units] {BEFORE|AFTER} clk_name [TIMEGRP  
group_name];
```

From-To (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 (FF、PAD、RAM) です。詳細は、『[制約ガイド](#)』の「FROM-TO」を参照してください。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能エレメント

詳細は、『[制約ガイド](#)』の「FROM-TO」を参照してください。

適用ルール

詳細は、『[制約ガイド](#)』の「FROM-TO」を参照してください。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

XCF の構文例

```
TIMESPEC TName = FROM group1 TO group2 value;
```

タイミング名 (TNM)

グループを構成するエレメントを指定する基本的なグループ制約で、作成したグループにタイミング仕様を設定できます。TNM は、特定の FFS、RAMS、LATCHES、PADS、BRAMS_PORTA、BRAMS_PORTB、CPUS、DSPS、HSIOS、MULTS をグループのエレメントとして指定して、そのグループに対するタイミング仕様の適用を簡略化するために使用します。

この制約は、キーワード RISING および FALLING と一緒に使用することもできます。詳細は、『[制約ガイド](#)』の「TNM」を参照してください。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能エレメント

詳細は、『[制約ガイド](#)』の「TNM」を参照してください。

適用ルール

詳細は、『[制約ガイド](#)』の「TNM」を参照してください。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

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 (TNM_NET)

[ネットのタイミング名 \(TNM_NET\)](#) 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。[タイミング名 \(TNM\)](#) および TNM_NET を DLL、DCM、および PLL で、PERIOD 制約と使用する場合は、特別なルールが適用されます。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。

TNM_NET は通常、特定のネットをグループ化するため HDL デザインで使用します。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、すべてグループと見なされます。詳細は、『[制約ガイド](#)』の「TNM_NET」を参照してください。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能エレメント

詳細は、『[制約ガイド](#)』の「TNM_NET」を参照してください。

適用ルール

詳細は、『[制約ガイド](#)』の「TNM_NET」を参照してください。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

XCF の構文例

```
NET netname TNM_NET = [predefined_group:] identifier;
```

タイムグループ (TIMEGRP)

TIMEGRP は、基本的なグループ制約です。TNM 識別子を使用したグループの作成に加え、ほかのグループに相対的にグループを指定できます。TIMEGRP 制約を使用すると、既存グループを組み合わせることでグループを作成できます。

TIMEGRP 制約は XST Constraint File (XCF) または Netlist Constraints File (NCF) に含めます。TIMEGRP を使用したグループの作成は、次のいずれかの方法で行います。

- ・ 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

詳細は、『[制約ガイド](#)』の「TIMEGRP」を参照してください。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能エレメント

詳細は、『[制約ガイド](#)』の「TIMEGRP」を参照してください。

適用ルール

詳細は、『[制約ガイド](#)』の「TIMEGRP」を参照してください。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

XCF の構文例

```
TIMEGRP newgroup = existing_grp1 existing_grp2 [existing_grp3 ...];
```

タイミング無視 (TIG)

TIG 制約を設定すると、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。この制約は、無視する信号の名前に適用します。詳細は、『[制約ガイド](#)』の「TIG」を参照してください。

アーキテクチャ サポート

Virtex®-6 および Spartan®-6 デバイスにのみ適用されます。

適用可能エレメント

詳細は、『[制約ガイド](#)』の「TIG」を参照してください。

適用ルール

詳細は、『[制約ガイド](#)』の「TIG」を参照してください。

構文例

次の例では、この制約を特定のツールまたは手法でどのように設定するかを示しています。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。

XCF の構文例

```
NET net_name TIG;
```


サードパーティの制約

メモ: 『XST ユーザー ガイド (Virtex-6 および Spartan-6 用)』は、Virtex®-6 および Spartan®-6 デバイス専用のマニュアルです。その他のデバイスを使用して XST を実行する場合は、『XST ユーザー ガイド』を参照してください。

この章では、次のセクションに分けて XST でサポートされるサードパーティ制約について説明します。

- ・ サードパーティの制約と同等の XST 制約
- ・ サードパーティ制約の構文例

サードパーティの制約と同等の XST 制約

サードパーティ制約の中には XST で自動的にサポートされるものもあります。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。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。

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。

VHDL では標準的な属性構文が使用されるので、HDL コードを変更する必要はありません。

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	Synplicity	ボックス タイプ	なし	VHDL Verilog
black_box_pad_pin	Synplicity	なし	なし	なし
black_box_tri_pins	Synplicity	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	Synplicity	フル ケース	なし	Verilog
	Synopsys			
ispad	Synplicity	なし	なし	なし
map_to_module	Synopsys	なし	なし	なし

名前	ベンダー	同等の XST 制約	自動認識	HDL
net_name	Synopsys	なし	なし	なし
parallel_case	Synplicity	パラレル ケース	なし	Verilog
	Synopsys			
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	Synplicity	レジスタ自動調整	なし	VHDL
				Verilog
syn_black_box	Synplicity	ボックス タイプ	あり	VHDL
				Verilog
syn_direct_enable	Synplicity	なし	なし	なし
syn_edif_bit_format	Synplicity	なし	なし	なし
syn_edif_scalar_format	Synplicity	なし	なし	なし
syn_encoding	Synplicity	FSM エンコード方法の指定	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	Synplicity	列挙型エンコード	なし	VHDL
syn_hier	Synplicity	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	Synplicity	なし	なし	なし
syn_keep	Synplicity	キープ	あり	VHDL
				Verilog

名前	ベンダー	同等の XST 制約	自動認識	HDL
syn_maxfan	Synplicity	最大ファンアウト	あり	VHDL
				Verilog
syn_netlist_hierarchy	Synplicity	ネットリスト階層	なし	VHDL
				Verilog
syn_noarrayports	Synplicity	なし	なし	なし
syn_noclockbuf	Synplicity	バッファ タイプ	あり	VHDL
				Verilog
syn_noprune	Synplicity	インスタンスエートされたプリミティブの最適化	あり	VHDL
				Verilog
syn_pipeline	Synplicity	レジスタ自動調整	なし	VHDL
				Verilog
syn_preserve	Synplicity	等価レジスタの削除	あり	VHDL
				Verilog
syn_ramstyle	Synplicity	RAM 抽出 および RAM スタイル	あり	VHDL
			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。	Verilog
			area 値は無視されます。	
syn_reference_clock	Synplicity	なし	なし	なし
syn_replicate	Synplicity	レジスタの複製	あり	VHDL
				Verilog
syn_romstyle	Synplicity	ROM 抽出 および ROM スタイル	あり	VHDL
				Verilog
syn_sharing	Synplicity	なし	なし	VHDL
				Verilog
syn_state_machine	Synplicity	FSM の自動抽出	あり	VHDL
				Verilog
syn_tco	Synplicity	なし	なし	なし
syn_tpd	Synplicity	なし	なし	なし
syn_tristate	Synplicity	なし	なし	なし
syn_tristatetomux	Synplicity	なし	なし	なし
syn_tsu	Synplicity	なし	なし	なし
syn_useenables	Synplicity	クロック イネーブルの使用	なし	なし
syn_useioff	Synplicity	I/O レジスタの IOB 内へのバック (IOB)	なし	VHDL
				Verilog

名前	ベンダー	同等の XST 制約	自動認識	HDL
synthesis_translate_off	Synplicity	Translate Off と Translate On	あり	VHDL
synthesis_translate_on	Synopsys			Verilog
xc_alias	Synplicity	なし	なし	なし
xc_clockbuftype	Synplicity	バッファ タイプ	なし	VHDL
				Verilog
xc_fast	Synplicity	FAST	なし	VHDL
				Verilog
xc_fast_auto	Synplicity	FAST	なし	VHDL
				Verilog
xc_global_buffers	Synplicity	BUFG (XST)	なし	VHDL
				Verilog
xc_ioff	Synplicity	I/O レジスタの IOB 内へのパック	なし	VHDL
				Verilog
xc_isgsr	Synplicity	なし	なし	なし
xc_loc	Synplicity	LOC	あり	VHDL
				Verilog
xc_map	Synplicity	単一 LUT へのエンティティのマップ	あり XST でサポートされる値は lut のみです	VHDL
				Verilog
xc_ncf_auto_relax	Synplicity	なし	なし	なし
xc_nodelay	Synplicity	NODELAY	なし	VHDL
				Verilog
xc_padtype	Synplicity	I/O 規格	なし	VHDL
				Verilog
xc_props	Synplicity	なし	なし	なし
xc_pullup	Synplicity	PULLUP	なし	VHDL
				Verilog
xc_rloc	Synplicity	RLOC	あり	VHDL
				Verilog
xc_fast	Synplicity	FAST	なし	VHDL
				Verilog
xc_slow	Synplicity	なし	なし	なし
xc_uset	Synplicity	U_SET	あり	VHDL
				Verilog

サードパーティ制約の構文例

このセクションでは、次のサードパーティ制約の構文例が含まれます。

- ・ Verilog 構文例
- ・ XCF の構文例

Verilog 構文例

```
module testkeep (in1, in2, out1);
  input in1;
  input in2;
  output out1;
  (* keep = "yes" *) wire aux1;
  (* keep = "yes" *) wire aux2;
  assign aux1 = in1;
  assign aux2 = in2;
  assign out1 = aux1 & aux2;
endmodule
```

XCF の構文例

キープ (KEEP) 制約は、別の合成制約ファイルでも設定できます。

```
BEGIN MODEL testkeep
```

```
NET aux1 KEEP=true;
```

```
END;
```

注意！XST Constraint File (XCF) 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、必ず二重引用符を使用する必要があります。

```
BEGIN MODEL testkeep
```

```
NET aux1 KEEP="soft";
```

```
END;
```

HDL デザインで信号/ネットを保持し、合成中に信号/ネットが最適化されないようにする方法は、この 2 つしかありません。

XST 合成レポート

メモ: 『XST ユーザー ガイド (Virtex-6 および Spartan-6 用)』は、Virtex®-6 および Spartan®-6 デバイス専用のマニュアルです。その他のデバイスを使用して XST を実行する場合は、『XST ユーザー ガイド』を参照してください。

この章では、XST の合成レポートについて次のセクションに分けて説明します。

- ・ [XST 合成レポートの概要](#)
- ・ [XST 合成レポートの内容](#)
- ・ [XST 合成レポートのナビゲーション](#)
- ・ [XST 合成レポートの情報](#)

XST 合成レポートの概要

XST 合成レポートは、

- ・ 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、XST 合成レポートを使用すると、

- ・ 合成の進捗状況を制御できます。
- ・ その時点までで実行されていることを確認できます。

合成後は、XST 合成レポートを使用すると、

- ・ HDL 記述が予測どおりに処理されたかどうかを判断できます。
- ・ 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見たしそうかどうかを判断できます。

XST 合成レポートの内容

XST 合成レポートには、次のセクションが含まれます。

- ・ [XST 合成レポートの目次](#)
- ・ [合成オプションのサマリ – Synthesis Options Summary セクション](#)
- ・ [XST 合成レポート – HDL Parsing および Elaboration セクション](#)
- ・ [XST 合成レポート – HDL Synthesis セクション](#)
- ・ [XST 合成レポート – Advanced HDL Synthesis セクション](#)
- ・ [XST 合成レポート – Low Level Synthesis セクション](#)
- ・ [XST 合成レポート – Partition Report セクション](#)
- ・ [XST 合成レポート – Design Summary セクション](#)

XST 合成レポートの目次

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、「[XST 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

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やオプションがまとめられています。

XST 合成レポート – HDL Parsing および Elaboration セクション

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
- ・ VHDL および Verilog ファイルの内容の解釈
- ・ デザイン階層の認識
- ・ HDL コードのミスを表示
- ・ 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。

XST 合成レポート – HDL Synthesis セクション

HDL 合成中、XST では次が実行されます。

- ・ 後で特定のインプリメンテーションが可能なレジスタ、加算器、乗算器などの基本的なマクロを認識します。
- ・ 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。

XST 合成レポート – Advanced HDL Synthesis セクション

アドバンス HDL 合成中、XST では次が実行されます。

- ・ HDL 合成段階で推論された基本マクロをカウンタ、パイプライン接続された乗算器、乗累算ファンクションなどの大型のマクロ ブロックにまとめようとします。
- ・ 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を表示します。

このセクションの最後には、デザイン全体で認識されたマクロのサマリがマクロのタイプ別にまとめられます。

このアドバンス HDL 合成セクションのレポートに関する詳細は、[第 7 章「HDLコーディング手法」](#)を参照してください。

XST 合成レポート – Low Level Synthesis セクション

この Low Level Synthesis セクションには、等価フリップフロップの削除、レジスタの複製、定数フリップフロップの最適化などを含む、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。

XST 合成レポート – Partition Report セクション

このセクションには、デザインがパーティション処理された場合の詳細なパーティション情報が含まれます。

XST 合成レポート – Design Summary セクション

この Design Summary セクションには、合成が問題なく終了したかどうか、特に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達成しているかどうか記述されます。

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

このサブセクションには、すべてのデバイス プリミティブと認識されたブラック ボックスの使用統計が表示されます。

プリミティブは、次のグループに分類されます。

- ・ **BELS**

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

このサブセクションには、次のようなファンクションの XST 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

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

このサブセクションには、XST の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

このサブセクションには、デザインのクロック数、各クロックがどのようにバッファを介しているか、およびそれに対応するファンアウトに関する情報が表示されます。

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

このサブセクションには、デザインの非同期セット/リセット数、各信号がどのようにバッファを介しているか、およびそれに対応するファンアウトに関する情報が表示されます。

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

この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 (レジスタから出力へのパス)
- ・ 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

このサブセクションには、次を含む各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価値のある情報など
- ・ 配線とロジック間の分配

Timing Details サブセクションの例

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

Cell:in->out	fanout	Gate Net		Logical Name (Net Name)
		Delay	Delay	
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)

暗号化されたモジュール

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。

XST 合成レポートのナビゲーション

このセクションでは、コマンドラインおよび ISE® Design Suite で合成レポートをナビゲーションする方法について説明します。

- ・ [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 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

コマンドラインモードを使用する場合、XST では SRP (`.srp`) ファイルが生成されます。SRP ファイルとは、次のようなファイルです。

- ・ XST 合成レポートすべてを含有
- ・ ASCII テキストファイル
- ・ テキストエディタで開くことが可能

SRP ファイルの目次には、リンクが付いていませんので、テキストエディタの検索機能を使用して該当箇所を探してください。

ISE Design Suite のレポートナビゲーション

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。

- ・ XST 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
- ・ XST 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

XST 合成レポートの情報

次を使用すると、XST 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](#)
- ・ [Quiet モード](#)
- ・ [Silent モード](#)

メッセージフィルタ

ISE® Design Suite で XST を実行する場合は、メッセージフィルタウィザードを使用して XST 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できます。詳細は、ISE Design Suite ヘルプの「メッセージフィルタの使用」を参照してください。

Quiet モード

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。XST 合成レポート自体の内容は変わりません。XST 合成レポートには、合成情報がすべてフィルタのか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。

- ・ **ise**

ISE® Design Suite 用にメッセージがフォーマットされます。

- ・ **xflow**

XFLOW 用にメッセージがフォーマットされます。

通常は、画面 (stdout) にすべてのログが出力されます。Quiet モードを使用して出力されなくなる 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タイミング レポートの注記
 - タイミングの詳細
 - CPU (XST のランタイム)
 - メモリ使用率

Quiet モードを使用して表示される 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](#)

Silent モード

Silent モードを使用すると、コンピュータ画面 (stdout) にはメッセージはまったく表示されませんが、ログ ファイルにはメッセージがすべて記述されます。このモードを設定するには、`-intstyle` オプションを `silent` にします。

XST の命名規則

メモ: 『XST ユーザー ガイド (Virtex-6 および Spartan-6 用)』は、Virtex®-6 および Spartan®-6 デバイス専用のマニュアルです。その他のデバイスを使用して XST を実行する場合は、『XST ユーザー ガイド』を参照してください。

この章では、XST の命名規則について次のセクションに分けて説明します。

- ・ [命名規則の概要](#)
- ・ [ネット命名規則のコード例](#)
- ・ [ネットの命名規則](#)
- ・ [インスタンスの命名規則](#)
- ・ [大文字/小文字の保持](#)
- ・ [命名規則の制御方法](#)

命名規則の概要

合成ツールは、命名方法に従って論理的で、一貫し、予測および繰り返し可能な名前をオブジェクトに付け、合成済みネットリストに書き込みます。XST の命名規則を使用することで、制約付きデザインのインプリメンテーションを制御したり、タイミング クロージャ サイクルを削減したりといった目標を達成しやすくなります。

XST 命名規則のコード例

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。

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. 信号名の階層を維持します。階層区切り文字は、[階層区切り文字](#)で定義したものが使用されます。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** の入力ネット名は、`<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、`<bus_name><left_delimiter><position>#<right_delimiter>` のような形式になります。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。

インスタンスの命名規則

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. インスタンス名の階層を維持します。階層区切り文字は、[階層区切り文字](#)で定義したものが使用されます。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 文があるとします。

```
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 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 の入力インスタンスには instance_name_OBUF .

大文字/小文字の保持

Verilog では、大文字と小文字が区別されます。case オプション (-case) で特に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。Verilog の大文字/小文字に関する XST のサポートおよびその制限については、[第 4 章「Verilog のサポート」](#)の「[大文字/小文字の区別](#)」を参照してください。

VHDL では大文字と小文字は区別されません。case オプション (-case) で特に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。

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。

- ・ [階層区切り文字の指定 \(-hierarchy_separator\)](#)
- ・ [バスの区切り文字指定 \(-bus_delimiter\)](#)
- ・ [大文字/小文字の指定\(-case\)](#)
- ・ [複製接尾語の設定 \(-duplication_suffix\)](#)

これらの制約は、ISE® Design Suite の [Synthesize - XST] プロセスのプロパティからか、該当するコマンドライン オプションで適用できます。

詳細は、[第 9 章「デザイン制約」](#)を参照してください。