

XST ユーザー ガイド

UG627 (v11.3) 2009 年 9月 16 日

本資料は英語版 (v11.3) を翻訳したものです。英語の更新バージョンがリリースされている場合には、最新の英語版を必ずご参照ください。

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.

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	17
XST ユーザー ガイドの内容	17
その他のリソース	17
表記規則	18
書体	18
オンライン マニュアル	18
2 : はじめに	21
XST の概要	21
11.1 の新機能	21
XST オプションの設定	21
3 : HDL コーディング手法	23
XSTでの符号のサポート	24
レジスタの HDL コーディング手法	24
レジスタのログ ファイル	25
レジスタ関連の制約	25
レジスタのコード例	25
ラッチの HDL コーディング手法	32
ラッチのログ ファイル	32
ラッチ関連の制約	32
ラッチのコード例	32
ポジティブ ゲート付きラッチの VHDL コード例	33
トライステートの HDL コーディング手法	36
トライステートのログ ファイル	36
トライステート関連の制約	36
トライステートのコード例	36
カウンタの HDL コーディング手法	38
カウンタのログ ファイル	39
カウンタ関連の制約	39
カウンタのコード例	39
アキュムレータの HDL コーディング手法	50
Virtex-4 および Virtex-5 デバイスのアキュムレータ	50
アキュムレータのログ ファイル	51
アキュムレータ関連の制約	51
アキュムレータのコード例	51
シフトレジスタの HDL コーディング手法	52
シフトレジスタの記述	53
シフトレジスタのインプリメンテーション	53
SRL16 および SRLC16	53
シフトレジスタのログ ファイル	55
シフトレジスタ関連の制約	55
シフトレジスタのコード例	55
ダイナミック シフトレジスタの HDL コーディング手法	67
ダイナミック シフトレジスタのログ ファイル	67
ダイナミック シフトレジスタ関連の制約	68
ダイナミック シフトレジスタのコード例	68
マルチプレクサの HDL コーディング手法	70
マルチプレクサの Verilog の [Case Implementation Style] パラメータ	71
マルチプレクサの Verilog の case 文のリソース	71
マルチプレクサのログ ファイル	72
マルチプレクサ関連の制約	72
マルチプレクサのコード例	72
デコーダの HDL コーディング手法	76
デコーダのログ ファイル	76
デコーダ関連の制約	77

デコーダのコード例	77
プライオリティ エンコーダの HDL コーディング手法	81
プライオリティ エンコーダのログ ファイル	81
プライオリティ エンコーダ関連の制約	81
プライオリティ エンコーダのコード例	81
1:9 の 3 ビットのプライオリティ エンコーダのコード例	82
論理シフタの HDL コーディング手法	83
論理シフタのログ ファイル	84
論理シフタ関連の制約	84
論理シフタのコード例	84
四則演算ブロックの HDL コーディング手法	88
四則演算ブロックのログ ファイル	89
四則演算ブロック関連の制約	89
加算器、減算器、加減算器の HDL コーディング手法	89
加算器、減算器、加減算器のログ ファイル	90
加算器、減算器、加減算器関連の制約	90
加算器、減算器、加減算器のコード例	90
キャリー アウト付き符号なし 8 ビット加算器	92
コンパレータの HDL コーディング手法	99
コンパレータのログ ファイル	99
コンパレータ関連の制約	99
コンパレータのコード例	99
乗算器の HDL コーディング手法	100
レジスタ付き乗算器	100
乗算器	101
定数との乗算	102
乗算器のログ ファイル	102
乗算器関連の制約	102
乗算器のコード例	102
逐次型複素乗算器の HDL コーディング手法	103
逐次型複素乗算器のログ ファイル	104
逐次型複素乗算器関連の制約	104
逐次型複素乗算器のコード例	104
パイプライン乗算器の HDL コーディング手法	106
パイプライン乗算器のログ ファイル	108
パイプライン乗算器関連の制約	108
パイプライン乗算器のコード例	108
パイプライン乗算器 (外部、単一レジスタ) の VHDL コード例	109
パイプライン乗算器 (外部、単一レジスタ) の Verilog コード例	110
乗算/加減算器の HDL コーディング手法	114
Virtex®-4 および Virtex-5 デバイスの乗算/加減算器のコーディング手法	114
乗算/加減算器のログ ファイル	115
乗算/加減算器関連の制約	116
乗算/加減算器のコード例	116
MAC の HDL コーディング手法	119
Virtex-4 および Virtex-5 デバイスの MAC	119
MAC のログ ファイル	120
MAC 関連の制約	120
MAC のコード例	120
除算器の HDL コーディング手法	124
除算器のログ ファイル	124
除算器関連の制約	124
除算器のコード例	124
リソース共有の HDL コーディング手法	125
リソース共有のログ ファイル	126
リソース共有関連の制約	126
リソース共有のコード例	126
RAM/ROM の HDL コーディング手法	128
RAM/ROM のログ ファイル	129

RAM/ROM 関連の制約	131
RAM/ROM のコード例	131
RAM の初期化のコード例	174
RAM を HDL コードで直接初期化する例	174
外部ファイルからの RAM の初期化	179
ブロック RAM の初期化 (外部データ ファイル)	180
ブロック RAM リソースを使用した ROM の HDL コーディング手法	182
ブロック RAM リソースを使用した ROM のログ ファイル	183
ブロック RAM リソースを使用した ROM 関連の制約	183
ブロック RAM リソースを使用した ROM のコード例	183
パイプライン化された分散 RAM の HDL コーディング手法	189
パイプライン化された分散 RAM のログ ファイル	191
パイプライン化された分散 RAM 関連の制約	191
パイプライン化された分散 RAM のコード例	192
Finite State Machine (FSM) の HDL コーディング手法	194
Finite State Machine (FSM) コンポーネントの記述	194
ステートレジスタ	195
次ステートの論理式	195
unreachable ステート	195
Finite State Machine (FSM) 出力	195
Finite State Machine (FSM) 入力	195
ステート エンコード手法	196
自動ステート エンコード	196
ワンホット ステート エンコード	196
グレイ ステート エンコード	196
コンパクト ステート エンコード	196
ジョンソン ステート エンコード	196
シーケンシャル ステート エンコード	196
Speed1 ステート エンコード	196
ユーザー ステート エンコード	197
RAM ベースの Finite State Machine (FSM) 合成	197
セーフ Finite State Machine (FSM) インプリメンテーション	197
Finite State Machine (FSM) ログ ファイル	198
Finite State Machine (FSM) 関連の制約	199
Finite State Machine (FSM) のコード例	200
ブラック ボックスの HDL コーディング手法	206
ブラック ボックスのログ ファイル	207
ブラック ボックス関連の制約	207
ブラック ボックスのコード例	207
4 : FPGA の最適化	209
FPGA 専用の合成オプション	210
マクロ生成	211
数値演算ファンクション	211
マクロ生成におけるロード可能ファンクション	211
マルチプレクサ	212
プライオリティ エンコーダ	212
デコーダ	212
シフトレジスタ	212
RAM	213
XST で使用されるプリミティブ	213
推論された RAM のインプリメンテーションの制御	213
ROM	214
DSP48 ブロック リソース	215
ブロック RAM へのロジックのマッピング	216
ブロック RAM へのロジックのマッピングのログ ファイル	216
ブロック RAM へのロジックのマッピングのコード例	217
フリップフロップのリタイミング	219
フリップフロップのリタイミングの制限	220
フリップフロップのリタイミングの制御方法	220

パーティション	220
エリア制約を設定した場合のスピード最適化.....	220
FPGA 最適化のレポート セクション.....	222
セル使用率レポート	222
BEL のセル使用率	222
フリップフロップとラッチのセル使用率	223
RAM のセル使用率	223
シフタのセル使用率	223
トライステートのセル使用率	223
クロック バッファのセル使用率.....	223
IO バッファのセル使用率.....	223
論理セル使用率.....	224
その他のセル使用率	224
タイミング レポート.....	224
タイミング レポートの例.....	225
タイミング レポートのタイミング サマリ セクション	226
タイミング レポートの Detail セクション	226
タイミング レポートの回路図.....	226
タイミング レポートのパスとポート	227
インプリメンテーション制約.....	227
FPGA デバイス プリミティブのサポート.....	227
属性を使用したプリミティブの生成	228
プリミティブとブラック ボックス.....	228
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.....	229
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	229
Verilog 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	229
プリミティブ インスタンス化のガイドライン	229
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	229
プリミティブ関連の制約	230
プリミティブのコード例.....	230
UniMacro ライブラリの使用	231
コアの処理	232
コアの処理の VHDL コード例	232
[Read Cores] のオン/オフ	232
INIT および RLOC の指定	233
LUT_MAP 制約を使用して INIT 値を渡すコード例.....	233
フリップフロップの INIT 値を指定するコード例.....	235
フリップフロップの INIT 値および RLOC 値を指定するコード例.....	236
XST での PCI フローの使用	238
ロジックとフリップフロップの複製の回避	238
コアの自動読み込み機能のオフ	238
5 : CPLD の最適化.....	239
CPLD 合成オプション	239
CPLD 合成でサポートされるデバイス	239
CPLD 合成オプションの設定	240
マクロ生成のインプリメンテーション	240
CPLD 合成のログ ファイルの解析	241
CPLD 合成の制約	242
CPLD 合成結果の向上	242
周波数の向上.....	243
大規模デザインのフィット.....	245
6 : デザイン制約	247
デザイン制約のリスト	248
一般制約	249
HDL 制約.....	250
FPGA 制約 (タイミング制約以外)	250
CPLD 制約 (タイミング以外).....	251
タイミング制約	252
インプリメンテーション制約	252

サードパーティの制約	252
グローバル制約とオプションの設定	252
合成オプション設定	253
Hardware Description Language (HDL) オプションの設定	253
FPGA デバイスの Hardware Description Language (HDL) オプションの設定	253
CPLD デバイスの Hardware Description Language (HDL) オプションの設定	254
ザイリンクス特有オプションの設定	255
FPGA デバイスのザイリンクス特有オプションの設定	255
CPLD デバイスのザイリンクス特有オプションの設定	255
その他の XST コマンドライン オプションの設定	256
カスタム コンパイル ファイル リスト	256
VHDL 属性の構文	256
Verilog-2001 の属性	257
Verilog-2001 属性の構文	257
Verilog-2001 の制限	258
Verilog-2001 のメタ コメント	258
XST Constraint File (XCF)	258
XST Constraint File (XCF) 構文とその使用	259
ネイティブ UCF 制約とそれ以外の UCF 制約	259
ネイティブ UCF 制約	259
ネイティブ以外の User Constraints File (UCF) 制約	260
XST Constraint File (XCF) の構文の制限	260
制約の優先順位	260
XST 固有の制約 (タイミング以外)	261
XST コマンドラインのみのオプション	268
XST タイミング オプション	272
XST タイミング オプション : [Process Properties] またはコマンドライン	272
XST タイミング オプション : XST Constraint File (XCF)	273
一般制約	274
-iobuf (I/O バッファの追加)	274
BOX_TYPE (ボックス タイプ)	275
アーキテクチャ サポート	276
適用可能エレメント	276
適用ルール	276
-bus_delimiter (バスの区切り文字指定)	276
-case (大文字/小文字の指定)	277
-vlgcase (Case 文のインプリメンテーション形式)	278
-define (Verilog マクロ)	278
-duplication_suffix (複製接尾語の設定)	279
FULL_CASE (フル ケース)	281
アーキテクチャ サポート	281
適用可能エレメント	281
適用ルール	281
-rtlview (RTL 回路図の生成)	282
-generics (ジェネリック)	282
-hierarchy_separator (階層区切り文字の指定)	283
IOSTANDARD (I/O 規格)	284
KEEP (キープ)	284
KEEP_HIERARCHY (階層の維持)	285
アーキテクチャ サポート	286
適用可能エレメント	286
適用ルール	286
-lso (ライブラリの検索順)	287
LOC	288
-netlist_hierarchy (ネットリスト階層)	288
OPT_LEVEL (最適化エフォート レベル)	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

OPT_MODE (最適化方法の指定).....	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
PARALLEL_CASE (パラレル ケース).....	290
アーキテクチャ サポート	290
適用可能エレメント.....	291
適用ルール.....	291
RLOC (RLOC).....	291
S (保存).....	291
-uc (合成制約ファイルの指定).....	292
TRANSLATE_OFF (変換なし) および TRANSLATE_ON (変換あり).....	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
-iuc (合成制約ファイルの使用).....	293
-vlgincdir (Verilog の 'include ディレクトリの指定).....	294
-verilog2001 (Verilog 2001 の指定).....	294
-xsthdpini (HDL ライブラリ マップ ファイル).....	295
-xsthdpdir (作業ディレクトリ).....	296
-xsthdpdir (作業ディレクトリ) の例.....	296
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
Hardware Description Language (HDL) 制約	298
FSM_EXTRACT (FSM 自動抽出).....	298
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
ENUM_ENCODING (列挙型エンコード手法).....	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.....	300
EQUIVALENT_REGISTER_REMOVAL (等価レジスタの削除).....	300
アーキテクチャ サポート	300
適用可能エレメント.....	300
適用ルール.....	300
FSM_ENCODING (FSM エンコード方法の指定).....	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
MUX_EXTRACT (MUX の抽出).....	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
REGISTER_POWERUP (レジスタ電源投入時の値).....	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
RESOURCE_SHARING (リソース共有)	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
SAFE_RECOVERY_STATE (セーフ リカバリ ステート).....	306
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
SAFE_IMPLEMENTATION (セーフ インプリメンテーション).....	307

アーキテクチャ サポート	307
適用可能エレメント.....	308
適用ルール.....	308
SIGNAL_ENCODING (信号のエンコード方法).....	308
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
FPGA 制約 (タイミング制約以外).....	310
ASYNC_TO_SYNC (非同期から同期への変換).....	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.....	312
AUTO_BRAM_PACKING (自動 BRAM パッキング).....	312
アーキテクチャ サポート	312
適用可能エレメント.....	312
適用ルール.....	312
BRAM_UTILIZATION_RATIO (BRAM 使用率).....	313
アーキテクチャ サポート	313
適用可能エレメント.....	313
適用ルール.....	313
BUFFER_TYPE (バッファ タイプ).....	314
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
BUFGCE (BUFGCE の抽出).....	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
-sd (コアの検索ディレクトリ).....	315
デコーダの抽出 (DECODER_EXTRACT).....	316
デコーダの抽出 (DECODER_EXTRACT) のアーキテクチャ サポート.....	316
デコーダの抽出 (DECODER_EXTRACT) の適用可能エレメント.....	316
デコーダの抽出 (DECODER_EXTRACT) の伝播規則	316
構文例.....	316
DSP_UTILIZATION_RATIO (DSP 使用率).....	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
FSM_STYLE (FSM スタイル).....	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
POWER (電力削減).....	320
アーキテクチャ サポート	320
適用可能エレメント.....	320
適用ルール.....	320
READ_CORES (コアの読み込み).....	321
アーキテクチャ サポート	321
適用可能エレメント.....	321
適用ルール.....	322
SHIFT_EXTRACT (論理シフタの抽出).....	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
LC (LUT の結合).....	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
BRAM_MAP (BRAM へのロジックのマッピング).....	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
MAX_FANOUT (最大ファンアウト数).....	326
アーキテクチャ サポート	326
適用可能エレメント.....	326
適用ルール.....	326
MOVE_FIRST_STAGE (最初のフリップフロップ ステージの移動).....	327
アーキテクチャ サポート	328
適用可能エレメント.....	328
適用ルール.....	329
MOVE_LAST_STAGE (最後のフリップフロップ ステージの移動).....	329
アーキテクチャ サポート	329
適用可能エレメント.....	330
適用ルール.....	330
MULT_STYLE (乗算器スタイル).....	330
アーキテクチャ サポート	331
適用可能エレメント.....	331
適用ルール.....	331
MUX_STYLE (MUX スタイル).....	332
アーキテクチャ サポート	332
適用可能エレメント.....	332
適用可能エレメント.....	332
-bufg (グローバル クロック バッファ数).....	333
-bufr (リージョン クロック バッファ数).....	334
OPTIMIZE_PRIMITIVES (インスタンス化されたプリミティブの最適化).....	335
アーキテクチャ サポート	335
適用可能エレメント.....	335
適用ルール.....	335
IOB (I/O レジスタの IOB 内へのパック).....	336
PRIORITY_EXTRACT (プライオリティ エンコーダの抽出).....	336
アーキテクチャ サポート	336
適用可能エレメント.....	336
適用ルール.....	337
RAM_EXTRACT (RAM の抽出).....	337
アーキテクチャ サポート	337
適用可能エレメント.....	338
適用ルール.....	338
RAM_STYLE (RAM スタイル).....	338
アーキテクチャ サポート	339
適用可能エレメント.....	339
適用ルール.....	339
REDUCE_CONTROL_SETS (制御セットの削減).....	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
REGISTER_BALANCING (レジスタの自動調整).....	341
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.....	344
REGISTER_DUPLICATION (レジスタの複製).....	344
アーキテクチャ サポート	345
適用可能エレメント.....	345
適用ルール.....	345
ROM_EXTRACT (ROM の抽出).....	346
アーキテクチャ サポート	346
適用可能エレメント.....	346
適用ルール.....	346
ROM_STYLE (ROM スタイル).....	347

アーキテクチャ サポート	347
適用可能エレメント.....	347
適用ルール	347
SHREG_EXTRACT (シフトレジスタの抽出).....	348
アーキテクチャ サポート	348
適用可能エレメント.....	349
適用ルール	349
-slice_packing (スライス パッキング).....	349
USELOWSKEWLINES (ロー スキュー ラインの使用).....	350
XOR_COLLAPSE (XOR コラプス)	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
Slice (LUT-FF Pairs) Utilization Ratio (スライス (LUT-FF ペア) 使用率)	351
アーキテクチャ サポート	351
適用可能エレメント.....	352
適用ルール	352
SLICE_UTILIZATION_RATIO_MAXMARGIN (スライス (LUT-FF ペア) 使用率の許容範囲).....	353
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
LUT_MAP (単一 LUT へのエンティティのマップ).....	354
アーキテクチャ サポート	355
適用可能エレメント.....	355
適用ルール	355
USE_CARRY_CHAIN (キャリーチェーンの使用)	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
TRISTATE2LOGIC (トライステートからロジックへの変換)	357
アーキテクチャ サポート	357
適用可能エレメント.....	357
適用ルール	357
USE_CLOCK_ENABLE (クロック イネーブルの使用).....	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
USE_SYNC_SET (同期セットの使用)	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
USE_SYNC_RESET (同期リセットの使用).....	361
アーキテクチャ サポート	362
適用可能エレメント.....	362
適用ルール	362
USE_DSP48 (DSP48 の使用)	363
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
CPLD 制約 (タイミング以外).....	365
-pld_ce (クロック イネーブル).....	365
DATA_GATE (データ ゲート)	366
アーキテクチャ サポート	366
-pld_mp (マクロの保持)	366
NOREDUCE (削減なし)	367
-wysiwyg (WYSIWYG)	367
アーキテクチャ サポート	367
適用可能エレメント.....	367
適用ルール	367

構文例.....	367
-pld_xp (XOR の保持).....	368
タイミング制約	369
タイミング制約の指定	369
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	369
UCF を使用したタイミング制約の指定	369
NGC ファイルへの制約の書き込み	370
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	370
-cross_clock_analysis (クロス クロック解析).....	370
-write_timing_constraints (タイミング制約の書き込み).....	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
CLOCK_SIGNAL (クロック信号).....	371
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
-glob_opt (グローバルな最適化目標).....	372
グローバル最適化のドメインの定義	373
XST Constraint File (XCF) のタイミング制約のサポート	373
PERIOD (周期)	374
OFFSET (オフセット).....	374
FROM-TO (From-To 制約)	374
TNM (タイミング名)	374
TNM_NET (ネットのタイミング名).....	375
TIMEGRP (タイムグループ).....	375
TIG (タイミング無視)	375
インプリメンテーション制約	375
インプリメンテーション制約の構文例.....	376
インプリメンテーション制約の XST Constraint File (XCF) 構文例	376
インプリメンテーション制約の VHDL 構文例	376
インプリメンテーション制約の Verilog 構文例.....	376
RLOC	376
NOREDUCE	376
PWR_MODE (電力モード).....	377
アーキテクチャ サポート	377
サードパーティの制約.....	377
サードパーティの制約と同等の XST 制約	377
サードパーティ制約の構文例.....	380
7: VHDL 言語のサポート.....	383
VHDL の IEEE サポート.....	384
VHDL の IEEE の競合	384
VHDL で LRM に準拠しない構文	384
サポートされる VHDL ファイル タイプ	384
書き込み関数を使用した VHDL のデバッグ コード例	386
VHDL で書き込み関数を使用してデバッグする場合のルール	388
VHDL のデータ型	388
使用できる VHDL のデータ型.....	388
列挙型.....	389
ユーザー定義の列挙型.....	389
ビット ベクタ型	389
整数型.....	389
定義済み型	389
STD_LOGIC_1164 IEEE 型.....	389
オーバーロード データ型.....	390
オーバーロード列挙型	390
オーバーロード ビット ベクタ型	390
オーバーロード整数型	390

オーバーロード STD_LOGIC_1164 IEEE 型	391
オーバーロード STD_LOGIC_ARITH IEEE 型	391
VHDL の多次元配列型	391
VHDL のレコード型	392
VHDL の初期値	393
VHDL のローカル リセット/グローバル リセット	393
VHDL のメモリ エLEMENTのデフォルト初期値の概要	394
VHDL のオブジェクト	394
VHDL の信号	394
VHDL の変数	394
VHDL の定数	395
VHDL の演算子	395
VHDL のエンティティとアーキテクチャの記述	396
VHDL の回路記述	396
VHDL のエンティティ宣言	396
VHDL のアーキテクチャ宣言	396
VHDL のコンポーネント インスタンス化	397
VHDL の再帰的なコンポーネント インスタンス化文	398
VHDL のコンポーネント コンフィギュレーション文	399
VHDL のジェネリック パラメータ宣言	400
VHDL のジェネリックと属性の競合	401
VHDL の組み合わせ回路	401
VHDL の同時処理信号代入文	401
VHDL のジェネレート文	402
VHDL の組み合わせプロセス文	403
if - else 文	405
VHDL の case 文	406
VHDL の for - loop 文	407
VHDL の順序回路	407
VHDL のセンシティビティ リスト付き順次プロセス文	407
VHDL のセンシティビティ リストのない順次プロセス文	408
レジスタおよびカウンタの VHDL コード例	408
VHDL での複数の wait 文の記述	410
VHDL の関数とプロシージャ	412
VHDL のアサート文	413
パッケージ文を使用した VHDL モデルの定義	415
標準パッケージ文を使用した VHDL モデルの定義	415
IEEE パッケージを使用した VHDL モデルの定義	416
Synopsys パッケージ文を使用した VHDL モデルの定義	417
サポートされる VHDL 構文	417
VHDL のデザイン エンティティとコンフィギュレーション	417
VHDL の論理式	419
VHDL 文	420
VHDL の予約語	421
8 : Verilog 言語のサポート	423
Verilog ビヘイビア記述	424
変数による部分的ビット選択	424
Verilog 構造記述	424
Verilog パラメータ	426
Verilog パラメータと属性の競合	427
Verilog パラメータと属性の優先順位	427
Verilog の制限	428
Verilog の大文字/小文字の区別	428
大文字/小文字に関するXST のサポート	428
XST 内での Verilogの制限	428
Verilog のブロッキングおよびノンブロッキング代入文	429
Verilog の整数処理	429
case 文での整数処理	430
連結文での整数処理	430

Verilog の属性とメタ コメント	430
Verilog-2001 の属性	430
Verilog のメタ コメント	431
サポートされる Verilog 構文	431
サポートされる Verilog の定数	432
サポートされる Verilog のデータ型	432
サポートされる Verilog の継続代入文	432
サポートされる Verilog の手続き代入文	432
サポートされる Verilog のデザイン階層	433
サポートされる Verilog のコンパイラ指示子	433
サポートされるシステム タスクと関数	434
Verilog プリミティブ	435
Verilog の予約言語	436
Verilog-2001 のサポート	437
9 : Verilog ビヘイビア記述のサポート	439
Verilog ビヘイビア記述の変数宣言	439
Verilog ビヘイビア記述の初期値	440
Verilog ビヘイビア記述のローカル リセット	440
Verilog ビヘイビア記述の配列のコード例	441
Verilog ビヘイビア記述の多次元配列	441
Verilog ビヘイビア記述のデータ型	442
Verilog ビヘイビア記述で使用可能な文	442
Verilog ビヘイビア記述の論理式	443
Verilog ビヘイビア記述でサポートされる演算子	443
Verilog ビヘイビア記述でサポートされる論理式	444
Verilog のビヘイビア記述の論理式の評価結果	445
Verilog ビヘイビア記述のブロック	446
Verilog ビヘイビア記述のモジュール	446
Verilog ビヘイビア記述のモジュール宣言	447
Verilog ビヘイビア記述の継続代入文	447
Verilog ビヘイビア記述の手続き代入文	448
Verilog ビヘイビア記述の組み合わせ always ブロック	448
Verilog ビヘイビア記述の if - else 文	448
Verilog ビヘイビア記述の case 文	449
Verilog ビヘイビア記述の for および repeat ループ文	449
Verilog ビヘイビア記述の While ループ	450
Verilog ビヘイビア記述の順次 always ブロック	451
Verilog ビヘイビア記述の assign 文および deassign 文	452
同じ always ブロックで実行される assign/deassign 文のコード例	453
assign/deassign 文で信号のビット/部分的ビット選択ができない場合のコード例	453
Verilog ビヘイビア記述の 32 ビットを超える場合のビットの拡張	454
Verilog ビヘイビア記述のタスクおよび関数	454
Verilog ビヘイビア記述の再帰タスクおよび関数	455
Verilog ビヘイビア記述の定数関数	455
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	456
Verilog ビヘイビア記述の定数	457
Verilog ビヘイビア記述のマクロ	457
Verilog ビヘイビア記述の include ファイル	457
Verilog ビヘイビア記述のコメント	458
Verilog ビヘイビア記述の generate 文	458
Verilog ビヘイビア記述の generate - for 文	459
Verilog ビヘイビア記述の generate - if - else 文	459
Verilog ビヘイビア記述の generate - case 文	459
10 : 混合言語のサポート	461
混合言語のプロジェクト ファイル	462
混合言語プロジェクトのVHDL/Verilog の境界規則	462
VHDL デザインへの Verilog モジュールのインスタンスシート	462
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	463
混合言語プロジェクトのポート マップ	464

Verilog 内の VHDL ポート マップ	464
VHDL 内の Verilog ポート マップ	464
混合言語内の VHDL ポート マップ	464
混合言語の Verilog ポート マップ	464
混合言語プロジェクトのジェネリック サポート	465
混合言語プロジェクトのライブラリ検索順 (LSO) ファイル	465
ISE Design Suite でのライブラリ検索順 (LSO) ファイルの指定	465
コマンドラインでのライブラリ検索順 (LSO) ファイルの指定	465
ライブラリ検索順 (LSO) に関する規則	465
LSO ファイルが空の場合	465
DEFAULT_SEARCH_ORDER キーワードのみの場合	466
DEFAULT_SEARCH_ORDER キーワードとライブラリリストがある場合	466
ライブラリリストのみの場合	467
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	467
11 : XST ログ ファイル	469
FPGA のログ ファイルの内容	469
FPGA ログ ファイルの著作権情報	469
FPGA のログ ファイルの目次	469
FPGA ログ ファイルの合成オプションのサマリ	470
FPGA ログ ファイルの HDL コンパイル	470
FPGA ログ ファイルのデザイン階層解析	470
FPGA ログ ファイルの HDL 解析	470
FPGA ログ ファイルの Hardware Description Language (HDL) 合成レポート	470
FPGA ログ ファイルのアドバンス HDL 合成レポート	470
FPGA ログ ファイルの下位レベルの合成	470
FPGA ログ ファイルのパーティション レポート	471
FPGA ログ ファイルの最終レポート	471
ログ ファイルの容量削減方法	471
メッセージフィルタの使用	472
Quiet モードの使用	472
Silent モードの使用	472
特定メッセージの非表示	472
hdl_level と hdl_and_low_levels に設定した場合に削除されるメッセージ	473
low_level または hdl_and_low_levels に設定した場合に削除されるメッセージ	473
ログ ファイルのマクロ	473
ログ ファイルの例	474
12 : XST の命名規則	499
ネットの命名規則	499
インスタンスの命名規則	499
命名規則の制御方法	500
13 : コマンドライン モード	501
コマンドライン モードからの XST の実行	501
コマンドライン モードのファイルの種類	501
コマンドライン モードの一時ファイル	502
コマンドライン モードでのスペースを含む名前	502
コマンドライン モードからの XST の起動	502
XST シェルを使用した場合	502
スクリプトファイルを使用した場合	502
XST スクリプトの設定	503
run コマンドを使用した場合	503
set コマンドを使用した場合	505
elaborate コマンドを使用した場合	506
コマンドライン モードを使用した VHDL デザインの合成	506
スクリプトモードでの XST の実行 (VHDL)	508
コマンドライン モードを使用した Verilog デザインの合成	509
スクリプトモードでの XST の実行 (Verilog)	510
コマンドライン モードを使用した混合言語デザインでの合成	511
スクリプトモードでの XST の実行 (混合言語)	512

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す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

XST ユーザー ガイドの内容

『XST ユーザー ガイド』には、次が含まれています。

- ・ 第 1 章「このマニュアルについて」：『XST ユーザー ガイド』の概要を示します。
- ・ 第 2 章「概要」：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。
- ・ 第 3 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 4 章「FPGA の最適化」：FPGA を最適化するための制約の使用法、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 5 章「CPLD の最適化」：CPLD の合成オプションおよびマクロ生成について説明します。
- ・ 第 6 章「デザイン制約」：XST のデザイン制約とその詳細について説明します。
- ・ 第 7 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 8 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 9 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 10 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 11 章「ログ ファイル」：XST のログ ファイルについて説明します。
- ・ 第 12 章「命名規則」：XST の命名規則について説明します。
- ・ 第 13 章「コマンド ライン モード」：コマンド ラインからの XST 実行方法を説明します。

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が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。	<code>lowpwr = {on off}</code>
縦棒	選択するリストの項目を分離します。	<code>lowpwr = {on off}</code>
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。	<code>allow block . . . block_name</code> <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。

はじめに

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。この章は、次のセクションから構成されています。

- ・ [XST の概要](#)
- ・ [11.1 の新機能](#)
- ・ [XST オプションの設定](#)

XST の概要

Xilinx Synthesis Technology (XST) は、HDL (ハードウェア記述言語) デザインを合成して NGC というザイリンクス用のネットリスト ファイルを生成するザイリンクスのアプリケーションです。NGC ファイルとは、論理デザイン データと制約の情報を含むネットリストで、Electronic Data Interchange Format (EDIF) および Netlist Constraints File (NCF) ファイル両方の代わりになります。

XST の詳細は、[Xilinx Synthesis Technology \(XST\) - よくある質問 \(FAQ\)](#)を参照してください。このページで XST FAQ というキーワードで検索してください。

11.1 の新機能

次に、11.1 から XST に追加された主な変更点を示します。

- ・ [KEEP](#) 制約に新しい値 `soft` を追加しました。この値を使用すると、XST でデザイン ネットが保持されますが、[KEEP](#) 制約は合成ネットリストには伝播されません。
- ・ `syn_keep` 制約が Synplify で処理される方法と矛盾しないように処理されるようになりました。`syn_keep` 制約を指定しても、XST では指定した信号が削除されないままになりますが、合成後のネットリストにこの制約は伝播されなくなっているため、配置配線が必要な場合にこのネットを最適化して削除して行うことができます。
- ・ XST では、インクリメンタル合成の代わりにパーティションがサポートされています。インクリメンタル合成はサポートされなくなりました。このため、`incremental_synthesis` および `resynthesize` 制約もサポートされません。パーティションの詳細は、ISE ヘルプを参照してください。

XST オプションの設定

デザインを合成する前に、XST 用のさまざまなオプションを設定できます。詳細は、次を参照してください。

- ・ [ISE® Design Suite ヘルプ](#)
- ・ [デザイン制約](#)
- ・ [コマンドライン モード](#)

デザインは通常、組み合わせロジックおよびマクロ (フリップフロップ、加算器、減算器、カウンタ、FSM、RAM など) から構成されています。マクロを使用すると、合成後のデザインのパフォーマンスが大きく向上します。そのため、マクロが XST で最適に処理されるようなコーディング手法を使用することが重要です。

XST は、まずできるだけ多くのマクロを認識 (推論) するよう試みます。この後、推論されたマクロに対して下位レベルの最適化が実行され、最適な結果が得られるよう個別のブロックとして保持されるか、周辺のロジックと結合されます。この処理は、マクロのタイプおよびビット幅によって異なります。たとえば 2:1 マルチプレクサは、デフォルトでは保持されません。合成制約を使用すると、推論されたマクロの処理を制御できます。詳細は、「[デザイン制約](#)」を参照してください。

HDL コーディング手法

この章には、デジタル ロジック回路のコーディング例が含まれます。この章は、次のセクションから構成されています。

- ・ XSTでの符号のサポート
- ・ レジスタの HDL コーディング手法
- ・ ラッチの HDL コーディング手法
- ・ トライステートの HDL コーディング手法
- ・ カウンタの HDL コーディング手法
- ・ アキュムレータの HDL コーディング手法
- ・ シフトレジスタの HDL コーディング手法
- ・ ダイナミック シフトレジスタの HDL コーディング手法
- ・ マルチプレクサの HDL コーディング手法
- ・ デコーダの HDL コーディング手法
- ・ プライオリティ エンコーダの HDL コーディング手法
- ・ 論理シフタの HDL コーディング手法
- ・ 四則演算ブロックの HDL コーディング手法
- ・ 加算器、減算器、加減算器の HDL コーディング手法
- ・ コンパレータの HDL コーディング手法
- ・ 乗算器の HDL コーディング手法
- ・ 逐次型複素乗算器の HDL コーディング手法
- ・ パイプライン乗算器の HDL コーディング手法
- ・ 乗算/加減算器の HDL コーディング手法
- ・ MAC の HDL コーディング手法
- ・ 除算器の HDL コーディング手法
- ・ リソース共有の HDL コーディング手法
- ・ RAM/ROM の HDL コーディング手法
- ・ パイプライン化された分散 RAM の HDL コーディング手法
- ・ 有限ステート マシン (FSM) の HDL コーディング手法
- ・ ブラック ボックスの HDL コーディング手法

ほとんどのセクションで、次の項目が含まれます。

- ・ マクロの一般的な説明
- ・ サンプル ログ ファイル
- ・ XST でマクロ処理を制御するために使用できる制約
- ・ 回路図、ピンの説明などを含む VHDL および Verilog コード例

詳細については、「[FPGA の最適化](#)」および「[CPLD の最適化](#)」を参照してください。

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。

XSTでの符号のサポート

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 で符号付きおよび符号なしの値を両方使用できるようにするには、次の手順で Verilog 2001 を有効にする必要があります。

- ・ ISE® Design Suite から [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] ページで [Verilog 2001] オプションを [Yes] にします。
- ・ -verilog2001 コマンドライン オプションを yes に設定します。

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できます。

符号なし加算器

パッケージ	タイプ
numeric_std	unsigned
std_logic_arith	unsigned
std_logic_unsigned	std_logic_vector

符号付き加算器を作成するには、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。

符号付き加算器

パッケージ	タイプ
numeric_std	signed
std_logic_arith	signed
std_logic_signed	std_logic_vector

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。

レジスタの HDL コーディング手法

XST では、次の制御信号付きのフリップフロップが認識されます。

- ・ 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

詳細は、「[INIT および RLOC の指定](#)」を参照してください。

レジスタのログ ファイル

XST ログ ファイルには、マクロの認識段階で認識されたフリップフロップのタイプおよびビット幅が示されます。

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

Synthesizing Unit <registers_5>.
  Related source file is "registers_5.vhd".
  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/Latches            : 4

=====
...
```

Virtex®-4 などの新しいデバイス ファミリのリリースに従い、XST で同じレジスタの別々のスライスがそれぞれ異なる方法で最適化されることがあるようになりました。たとえば、あるレジスタの一部は DSP48 ブロックに挿入され、別の部分はスライスまたはシフトレジスタの一部としてインプリメントされる場合などです。そのため、XST ではアドバンス HDL 合成段階後に生成される HDL 合成レポートにデザインのフリップフロップの合計ビット数が示されるようになりました。

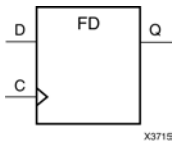
レジスタ関連の制約

- ・ [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 \(REGISTER_BALANCING\)](#)

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。アップデートは、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

クロックの立ち上がりエッジで動作するフリップフロップの図



クロックの立ち上がりエッジで動作するフリップフロップのピンの説明

I/O ピン	説明
D	データ入力
C	クロック (立ち上がりエッジ)
Q	データ出力

クロックの立ち上がりエッジで動作するフリップフロップの VHDL コード例

```
--
-- Flip-Flop with Positive-Edge Clock
--
library ieee;
use ieee.std_logic_1164.all;

entity registers_1 is
    port(C, D : in std_logic;
         Q   : out std_logic);
end registers_1;

architecture archi of registers_1 is
begin

    process (C)
    begin
        if (C'event and C='1') then
            Q <= D;
        end if;
    end process;

end archi;
```

VHDL を使用してクロックの立ち上がりエッジを記述する場合、

```
if (C'event and C='1') then
```

の代わりに、次の記述も使用できます。

```
if (rising_edge(C)) then
```

クロックの立ち上がりエッジで動作するフリップフロップの Verilog コード例

```
//
// Flip-Flop with Positive-Edge Clock
//

module v_registers_1 (C, D, Q);
    input  C, D;
    output Q;
    reg    Q;

    always @(posedge C)
    begin
        Q <= D;
    end

endmodule
```

クロックの立ち上がりエッジで動作するフリップ セットの INITSTATE 付きフリップフロップ

Verilog コード例

```
module test(d, C, q);
    input d;
    input C;
    output q;

    reg qtemp = 'b1 ;

    always @ (posedge C)
    begin
        qtemp = d;
    end

    assign q = qtemp;
endmodule
```

VHDL コード例

```
library ieee;
use ieee.std_logic_1164.all;

entity registers_1 is
    port(C, D : in std_logic;
         Q : out std_logic);
end registers_1;

architecture archi of registers_1 is
    signal qtemp : std_logic := '1';

begin

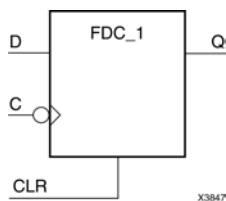
    process (C)

    begin
        if (C'event and C='1') then
            qtemp <= D;

        end if;
        Q <= Qtemp;
    end process;

end archi;
```

クロックの立ち下がリエッジで動作する非同期リセット付きフリップフロップの図



クロックの立ち下がリエッジで動作する非同期リセット付きフリップフロップのピンの説明

I/O ピン	説明
D	データ入力
C	クロック (立ち下がリエッジ)
CLR	非同期リセット (アクティブ High)
Q	データ出力

クロックの立ち下がリエッジで動作する非同期リセット付きフリップフロップの VHDL コード例

```
--
-- Flip-Flop with Negative-Edge Clock and Asynchronous Reset
--

library ieee;
use ieee.std_logic_1164.all;

entity registers_2 is
    port(C, D, CLR : in  std_logic;
         Q          : out std_logic);
end registers_2;

architecture archi of registers_2 is
begin

    process (C, CLR)
    begin
        if (CLR = '1')then
            Q <= '0';
        elsif (C'event and C='0')then
            Q <= D;
        end if;
    end process;

end archi;
```

クロックの立ち下がリエッジで動作する非同期リセット付きフリップフロップの Verilog コード例

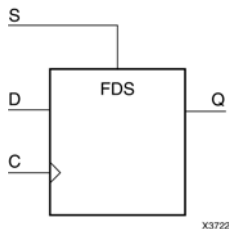
```
//
// Flip-Flop with Negative-Edge Clock and Asynchronous Reset
//

module v_registers_2 (C, D, CLR, Q);
    input  C, D, CLR;
    output Q;
    reg    Q;

    always @(negedge C or posedge CLR)
    begin
        if (CLR)
            Q <= 1'b0;
        else
            Q <= D;
        end
    end

endmodule
```

クロックの立ち上がりエッジで動作する同期セット付きフリップフロップの図



クロックの立ち上がりエッジで動作する同期セット付きフリップフロップのピンの説明

I/O ピン	説明
D	データ入力
C	クロック (立ち上がりエッジ)
S	同期セット (アクティブ High)
Q	データ出力

クロックの立ち上がりエッジで動作する同期セット付きフリップフロップの VHDL コード例

```
--
-- Flip-Flop with Positive-Edge Clock and Synchronous Set
--
library ieee;
use ieee.std_logic_1164.all;
entity registers_3 is
  port(C, D, S : in std_logic;
        Q : out std_logic);
end registers_3;

architecture archi of registers_3 is
begin

  process (C)
  begin
    if (C'event and C='1') then
      if (S='1') then
        Q <= '1';
      else
        Q <= D;
      end if;
    end if;
  end process;

end archi;
```

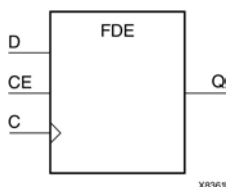
クロックの立ち上がりエッジで動作する同期セット付きフリップフロップの Verilog コード例

```
//
// Flip-Flop with Positive-Edge Clock and Synchronous Set
//
module v_registers_3 (C, D, S, Q);
  input C, D, S;
  output Q;
  reg Q;

  always @(posedge C)
  begin
    if (S)
      Q <= 1'b1;
    else
      Q <= D;
    end
  end

endmodule
```

クロックの立ち上がりエッジで動作するクロック イネーブル付きフリップフロップの図



クロックの立ち上がりエッジで動作するクロック イネーブル付きフリップフロップのピンの説明

I/O ピン	説明
D	データ入力
C	クロック (立ち上がりエッジ)
CE	クロック イネーブル (アクティブ High)
Q	データ出力

クロックの立ち上がりエッジで動作するクロック イネーブル付きフリップフロップの VHDL コード例

```
--
-- Flip-Flop with Positive-Edge Clock and Clock Enable
--
library ieee;
use ieee.std_logic_1164.all;

entity registers_4 is
    port(C, D, CE : in std_logic;
         Q       : out std_logic);
end registers_4;

architecture archi of registers_4 is
begin

    process (C)
    begin
        if (C'event and C='1') then
            if (CE='1') then
                Q <= D;
            end if;
        end if;
    end process;

end archi;
```

クロックの立ち上がりエッジで動作するクロック イネーブル付きフリップフロップの Verilog コード例

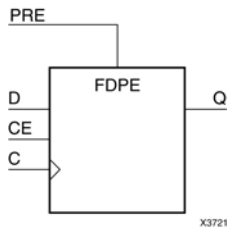
```
//
// Flip-Flop with Positive-Edge Clock and Clock Enable
//

module v_registers_4 (C, D, CE, Q);
    input  C, D, CE;
    output Q;
    reg    Q;

    always @(posedge C)
    begin
        if (CE)
            Q <= D;
    end

endmodule
```

クロックの立ち上がりエッジで動作する非同期セットおよびクロック イネーブル付き 4 ビットレジスタの図



クロックの立ち上がりエッジで動作する非同期セットおよびクロック イネーブル付き 4 ビットレジスタのピンの説明

I/O ピン	説明
D	データ入力
C	クロック (立ち上がりエッジ)
PRE	非同期セット (アクティブ High)
CE	クロック イネーブル (アクティブ High)
Q	データ出力

クロックの立ち上がりエッジで動作する非同期セットおよびクロック イネーブル付き 4 ビットレジスタの VHDL コード例

```
--
-- 4-bit Register with Positive-Edge Clock, Asynchronous Set and Clock Enable
--

library ieee;
use ieee.std_logic_1164.all;

entity registers_5 is
    port(C, CE, PRE : in std_logic;
         D          : in std_logic_vector (3 downto 0);
         Q          : out std_logic_vector (3 downto 0));
end registers_5;

architecture archi of registers_5 is
begin
    process (C, PRE)
    begin
        if (PRE='1') then
            Q <= "1111";
        elsif (C'event and C='1') then
            if (CE='1') then
                Q <= D;
            end if;
        end if;
    end process;
end archi;
```

クロックの立ち上がりエッジで動作する非同期セットおよびクロック イネーブル付き 4 ビット レジスタの Verilog コード例

```
//  
// 4-bit Register with Positive-Edge Clock, Asynchronous Set and Clock Enable  
//  
module v_registers_5 (C, D, CE, PRE, Q);  
    input  C, CE, PRE;  
    input  [3:0] D;  
    output [3:0] Q;  
    reg    [3:0] Q;  
  
    always @(posedge C or posedge PRE)  
    begin  
        if (PRE)  
            Q <= 4'b1111;  
        else  
            if (CE)  
                Q <= D;  
        end  
    end  
endmodule
```

ラッチの HDL コーディング手法

XST では、非同期セット/リセット付きのラッチが認識されます。ラッチは次の方法で記述できます。

- ・ process 文 (VHDL)
- ・ always ブロック (Verilog)
- ・ 同時処理ステート代入文

ラッチの記述では wait 文 (VHDL) はサポートされません。

ラッチのログ ファイル

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。

```
...  
Synthesizing Unit <latch>.  
    Related source file is latch_1.vhd.  
WARNING:Xst:737 - Found 1-bit latch for signal <q>.  
    Summary:  
        inferred    1 Latch(s).  
Unit <latch> synthesized.  
  
=====  
HDL Synthesis Report  
  
Macro Statistics  
# Latches                : 1  
  1-bit latch            : 1  
=====
```

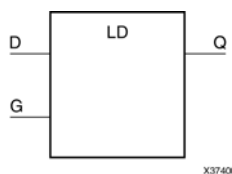
ラッチ関連の制約

[I/O レジスタの IOB 内へのパック \(IOB\)](#)

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。アップデートは、
ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

ポジティブ ゲート付きラッチの図



ポジティブ ゲート付きラッチのピンの説明

I/O ピン	説明
D	データ入力
G	ポジティブ ゲート
Q	データ出力

ポジティブ ゲート付きラッチの VHDL コード例

```
--
-- Latch with Positive Gate
--
library ieee;
use ieee.std_logic_1164.all;
entity latches_1 is
  port(G, D : in std_logic;
        Q : out std_logic);
end latches_1;

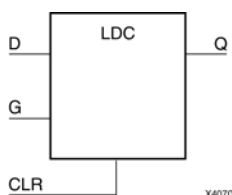
architecture archi of latches_1 is
begin
  process (G, D)
  begin
    if (G='1') then
      Q <= D;
    end if;
  end process;
end archi;
```

ポジティブ ゲート付きラッチの Verilog コード例

```
//
// Latch with Positive Gate
//
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
```

ポジティブ ゲートおよび非同期リセット付きラッチの図



ポジティブ ゲートおよび非同期リセット付きラッチのピンの説明

I/O ピン	説明
D	データ入力
G	ポジティブ ゲート
CLR	非同期リセット (アクティブ High)
Q	データ出力

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
--

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;
```

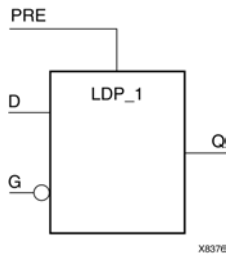
ポジティブ ゲートおよび非同期リセット付きラッチの Verilog コード例

```
//
// Latch with Positive Gate and Asynchronous Reset
//

module v_latches_2 (G, D, CLR, Q);
    input G, D, CLR;
    output Q;
    reg Q;

    always @(G or D or CLR)
    begin
        if (CLR)
            Q = 1'b0;
        else if (G)
            Q = D;
    end
endmodule
```

反転ゲートおよび非同期セット付き 4 ビット ラッチの図



反転ゲートおよび非同期セット付き 4 ビット ラッチのピンの説明

I/O ピン	説明
D	データ入力
G	反転ゲート
PRE	非同期プリセット (アクティブ High)
Q	データ出力

反転ゲートおよび非同期セット付き 4 ビット ラッチの VHDL コード例

```
--
-- 4-bit Latch with Inverted Gate and Asynchronous Set
--

library ieee;
use ieee.std_logic_1164.all;

entity latches_3 is
    port(D      : in std_logic_vector(3 downto 0);
          G, PRE : in std_logic;
          Q      : out std_logic_vector(3 downto 0));
end latches_3;

architecture archi of latches_3 is
begin
    process (PRE, G, D)
    begin
        if (PRE='1') then
            Q <= "1111";
        elsif (G='0') then
            Q <= D;
        end if;
    end process;
end archi;
```

反転ゲートおよび非同期セット付き 4 ビット ラッチの Verilog コード例

```
//
// 4-bit Latch with Inverted Gate and Asynchronous Set
//

module v_latches_3 (G, D, PRE, Q);
    input G, PRE;
    input [3:0] D;
    output [3:0] Q;
    reg [3:0] Q;

    always @(G or D or PRE)
    begin
        if (PRE)
            Q = 4'b1111;
        else if (~G)
            Q = D;
    end
endmodule
```

トリステートの HDL コーディング手法

トリステート エLEMENTは、次を使用して記述できます。

- ・ 組み合わせプロセス文 (VHDL)
- ・ always ブロック (Verilog)
- ・ 同時処理代入文

「トリステートのコード例」では、1 の代わりに 0 と比較することで、BUFE マクロではなく BUFT プリミティブが推論されています。BUFE マクロでは、E ピンにインバータが付いています。

トリステートのログ ファイル

XST ログ ファイルには、認識されたトリステート エLEMENTのタイプおよびビット幅が示されます。

```
...
Synthesizing Unit <three_st>.
  Related source file is tristates_1.vhd.
  Found 1-bit tristate buffer for signal <o>.
  Summary:
    inferred    1 Tristate(s).
Unit <three_st> synthesized.

=====
HDL Synthesis Report

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

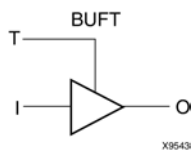
トリステート関連の制約

トリステートからロジックへの変換 (TRISTATE2LOGIC)

トリ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。アップデートは、
http://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

組み合わせプロセスおよび always ブロックを使用したトライステートの図



組み合わせプロセスおよび always ブロックを使用したトライステートのピンの説明

I/O ピン	説明
I	データ入力
T	出力イネーブル (アクティブ Low)
O	データ出力

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
--

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;
```

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

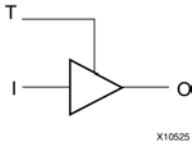
```
//
// Tristate Description Using Combinatorial Always Block
//

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
```

同時処理代入文を使用したトライステートの図



同時処理代入文を使用したトライステートのピンの説明

I/O ピン	説明
I	データ入力
T	出力イネーブル (アクティブ Low)
O	データ出力

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
--

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;
```

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
//

module v_three_st_2 (T, I, O);
    input  T, I;
    output O;

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

カウンタの HDL コーディング手法

XST では、次の制御信号付きのカウンタが認識されます。

- ・ 非同期セット/リセット
- ・ 同期セット/リセット
- ・ 非同期/同期ロード (信号または定数のいずれか、またはその両方)
- ・ クロック イネーブル
- ・ モード (アップ、ダウン、アップ/ダウン)
- ・ 上記の制御信号の組み合わせ

次の制御信号の HDL コーディング スタイルは、「レジスタの HDL コーディング手法」に記述されているものと同じです。

- ・ クロック
- ・ 非同期セット/リセット
- ・ 同期セット/リセット

XST では、符号付きおよび符号なし両方のカウンタがサポートされています。

カウンタのログ ファイル

XST ログ ファイルには、認識されたカウンタのタイプおよびビット幅が示されます。

```
...
Synthesizing Unit <counter>.
  Related source file is counters_1.vhd.
  Found 4-bit up counter for signal <tmp>.
  Summary:
    inferred    1 Counter(s).
  Unit <counter> synthesized.

=====
HDL Synthesis Report

Macro Statistics
# Counters                : 1
 4-bit up counter         : 1
=====
...
```

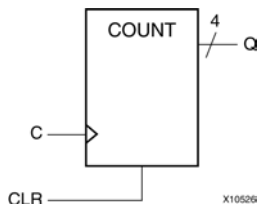
カウンタ関連の制約

- ・ DSP48 の使用 (USE_DSP48)
- ・ DSP 使用率 (DSP_UTILIZATION_RATIO)
- ・ キープ (KEEP)

カウンタのコード例

コード例は、本書が作成された時点のものです。アップデートは、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

非同期リセット付き 4 ビット符号なしアップ カウンタの図



非同期リセット付き 4 ビット符号なしアップ カウンタのピンの説明

I/O ピン	説明
C	クロック (立ち上がりエッジ)
CLR	非同期リセット (アクティブ High)
Q	データ出力

非同期リセット付き 4 ビット符号なしアップ カウンタの VHDL コード例

```
--
-- 4-bit unsigned up counter with an asynchronous reset.
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counters_1 is
    port(C, CLR : in std_logic;
         Q : out std_logic_vector(3 downto 0));
end counters_1;

architecture archi of counters_1 is
    signal tmp: std_logic_vector(3 downto 0);
begin
    process (C, CLR)
    begin
        if (CLR='1') then
            tmp <= "0000";
        elsif (C'event and C='1') then
            tmp <= tmp + 1;
        end if;
    end process;

    Q <= tmp;
end archi;
```

非同期リセット付き 4 ビット符号なしアップ カウンタの Verilog コード例

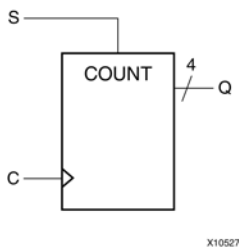
```
//
// 4-bit unsigned up counter with an asynchronous reset.
//

module v_counters_1 (C, CLR, Q);
    input C, CLR;
    output [3:0] Q;
    reg [3:0] tmp;

    always @(posedge C or posedge CLR)
    begin
        if (CLR)
            tmp <= 4'b0000;
        else
            tmp <= tmp + 1'b1;
        end

    assign Q = tmp;
endmodule
```

同期セット付き 4 ビット符号なしダウン カウンタの図



同期セット付き 4 ビット符号なしダウン カウンタのピンの説明

I/O ピン	説明
C	クロック (立ち上がりエッジ)
S	同期セット (アクティブ High)
Q	データ出力

同期セット付き 4 ビット符号なしダウン カウンタの VHDL コード例

```
--
-- 4-bit unsigned down counter with a synchronous set.
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counters_2 is
    port(C, S : in std_logic;
         Q : out std_logic_vector(3 downto 0));
end counters_2;

architecture archi of counters_2 is
    signal tmp: std_logic_vector(3 downto 0);
begin
    process (C)
    begin
        if (C'event and C='1') then
            if (S='1') then
                tmp <= "1111";
            else
                tmp <= tmp - 1;
            end if;
        end if;
    end process;

    Q <= tmp;
end archi;
```

同期セット付き 4 ビット符号なしダウン カウンタの Verilog コード例

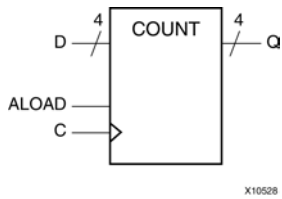
```
//
// 4-bit unsigned down counter with a synchronous set.
//

module v_counters_2 (C, S, Q);
    input C, S;
    output [3:0] Q;
    reg [3:0] tmp;

    always @(posedge C)
    begin
        if (S)
            tmp <= 4'b1111;
        else
            tmp <= tmp - 1'b1;
        end

        assign Q = tmp;
    endmodule
```

プライマリ入力からの非同期ロード付き 4 ビット符号なしアップ カウンタの図



プライマリ入力からの非同期ロード付き 4 ビット符号なしアップ カウンタのピンの説明

I/O ピン	説明
C	クロック (立ち上がりエッジ)
ALOAD	非同期ロード (アクティブ High)
D	データ入力
Q	データ出力

プライマリ入力からの非同期ロード付き 4 ビット符号なしアップ カウンタの VHDL コード例

```
--
-- 4-bit Unsigned Up Counter with Asynchronous Load from Primary Input
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counters_3 is
    port(C, ALOAD : in std_logic;
         D : in std_logic_vector(3 downto 0);
         Q : out std_logic_vector(3 downto 0));
end counters_3;

architecture archi of counters_3 is
    signal tmp: std_logic_vector(3 downto 0);
begin
    process (C, ALOAD, D)
    begin
        if (ALOAD='1') then
            tmp <= D;
        elsif (C'event and C='1') then
            tmp <= tmp + 1;
        end if;
    end process;

    Q <= tmp;
end archi;
```

プライマリ入力からの非同期ロード付き 4 ビット符号なしアップ カウンタの Verilog コード例

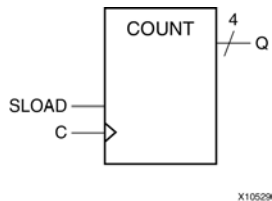
```
//
// 4-bit Unsigned Up Counter with Asynchronous Load from Primary Input
//

module v_counters_3 (C, ALOAD, D, Q);
    input C, ALOAD;
    input [3:0] D;
    output [3:0] Q;
    reg [3:0] tmp;

    always @(posedge C or posedge ALOAD)
    begin
        if (ALOAD)
            tmp <= D;
        else
            tmp <= tmp + 1'b1;
        end

    assign Q = tmp;
endmodule
```

定数の同期ロード付き 4 ビット符号なしアップ カウンタの図



定数の同期ロード付き 4 ビット符号なしアップ カウンタのピンの説明

I/O ピン	説明
C	クロック (立ち上がりエッジ)
SLOAD	同期ロード (アクティブ High)
Q	データ出力

定数の同期ロード付き 4 ビット符号なしアップ カウンタの VHDL コード例

```
--
-- 4-bit Unsigned Up Counter with Synchronous Load with a Constant
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counters_4 is
    port(C, SLOAD : in std_logic;
         Q : out std_logic_vector(3 downto 0));
end counters_4;

architecture archi of counters_4 is
    signal tmp: std_logic_vector(3 downto 0);
begin
    process (C)
    begin
        if (C'event and C='1') then
            if (SLOAD='1') then
                tmp <= "1010";
            else
                tmp <= tmp + 1;
            end if;
        end if;
    end process;

    Q <= tmp;

end archi;
```

定数の同期ロード付き 4 ビット符号なしアップ カウンタの Verilog コード例

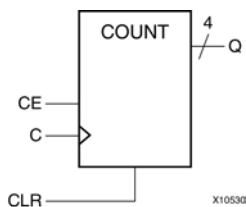
```
//
// 4-bit Unsigned Up Counter with Synchronous Load with a Constant
//

module v_counters_4 (C, SLOAD, Q);
    input C, SLOAD;
    output [3:0] Q;
    reg [3:0] tmp;

    always @(posedge C)
    begin
        if (SLOAD)
            tmp <= 4'b1010;
        else
            tmp <= tmp + 1'b1;
        end

        assign Q = tmp;
    endmodule
```

非同期リセットおよびクロック イネーブル付き 4 ビット符号なしアップ カウンタの図



非同期リセットおよびクロック イネーブル付き 4 ビット符号なしアップ カウンタのピンの説明

I/O ピン	説明
C	クロック (立ち上がりエッジ)
CLR	非同期リセット (アクティブ High)
CE	クロック イネーブル
Q	データ出力

非同期リセットおよびクロック イネーブル付き 4 ビット符号なしアップ カウンタの VHDL コード例

```
--
-- 4-bit Unsigned Up Counter with Asynchronous Reset and Clock Enable
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counters_5 is
    port(C, CLR, CE : in std_logic;
         Q : out std_logic_vector(3 downto 0));
end counters_5;

architecture archi of counters_5 is
    signal tmp: std_logic_vector(3 downto 0);
begin
    process (C, CLR)
    begin
        if (CLR='1') then
            tmp <= "0000";
        elsif (C'event and C='1') then
            if (CE='1') then
                tmp <= tmp + 1;
            end if;
        end if;
    end process;

    Q <= tmp;
end archi;
```

非同期リセットおよびクロック イネーブル付き 4 ビット符号なしアップ カウンタの Verilog コード例

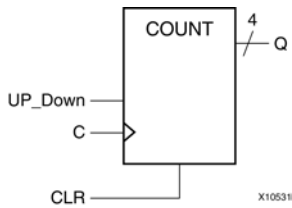
```
//
// 4-bit Unsigned Up Counter with Asynchronous Reset and Clock Enable
//

module v_counters_5 (C, CLR, CE, Q);
    input C, CLR, CE;
    output [3:0] Q;
    reg [3:0] tmp;

    always @(posedge C or posedge CLR)
    begin
        if (CLR)
            tmp <= 4'b0000;
        else if (CE)
            tmp <= tmp + 1'b1;
        end

        assign Q = tmp;
    endmodule
```

非同期リセット付き 4 ビット符号なしアップ/ダウン カウンタの図



非同期リセット付き 4 ビット符号なしアップ/ダウン カウンタのピンの説明

I/O ピン	説明
C	クロック (立ち上がりエッジ)
CLR	非同期リセット (アクティブ High)
UP_DOWN	アップ/ダウン カウント モード セレクタ
Q	データ出力

非同期リセット付き 4 ビット符号なしアップ/ダウン カウンタの VHDL コード例

```
--
-- 4-bit Unsigned Up/Down counter with Asynchronous Reset
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counters_6 is
    port(C, CLR, UP_DOWN : in std_logic;
         Q : out std_logic_vector(3 downto 0));
end counters_6;

architecture archi of counters_6 is
    signal tmp: std_logic_vector(3 downto 0);
begin
    process (C, CLR)
    begin
        if (CLR='1') then
            tmp <= "0000";
        elsif (C'event and C='1') then
            if (UP_DOWN='1') then
                tmp <= tmp + 1;
            else
                tmp <= tmp - 1;
            end if;
        end if;
    end process;

    Q <= tmp;
end archi;
```

非同期リセット付き 4 ビット符号なしアップ/ダウン カウンタの Verilog コード例

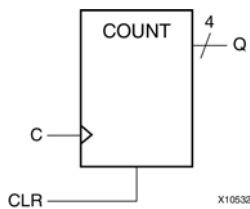
```
//
// 4-bit Unsigned Up/Down counter with Asynchronous Reset
//

module v_counters_6 (C, CLR, UP_DOWN, Q);
    input C, CLR, UP_DOWN;
    output [3:0] Q;
    reg [3:0] tmp;

    always @(posedge C or posedge CLR)
    begin
        if (CLR)
            tmp <= 4'b0000;
        else if (UP_DOWN)
            tmp <= tmp + 1'b1;
        else
            tmp <= tmp - 1'b1;
    end

    assign Q = tmp;
endmodule
```

非同期リセット付き 4 ビット符号付きアップ カウンタの図



非同期リセット付き 4 ビット符号付きアップ カウンタのピンの説明

I/O ピン	説明
C	クロック (立ち上がりエッジ)
CLR	非同期リセット (アクティブ High)
Q	データ出力

非同期リセット付き 4 ビット符号付きアップ カウンタの VHDL コード例

```
--
-- 4-bit Signed Up Counter with Asynchronous Reset
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity counters_7 is
    port(C, CLR : in std_logic;
         Q : out std_logic_vector(3 downto 0));
end counters_7;

architecture archi of counters_7 is
    signal tmp: std_logic_vector(3 downto 0);
begin
    process (C, CLR)
    begin
        if (CLR='1') then
            tmp <= "0000";
        elsif (C'event and C='1') then
            tmp <= tmp + 1;
        end if;
    end process;

    Q <= tmp;
end archi;
```

非同期リセット付き 4 ビット符号付きアップ カウンタの Verilog コード例

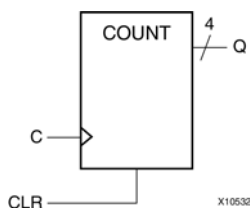
```
//
// 4-bit Signed Up Counter with Asynchronous Reset
//

module v_counters_7 (C, CLR, Q);
    input C, CLR;
    output signed [3:0] Q;
    reg signed [3:0] tmp;

    always @ (posedge C or posedge CLR)
    begin
        if (CLR)
            tmp <= 4'b0000;
        else
            tmp <= tmp + 1'b1;
        end

        assign Q = tmp;
    endmodule
```

非同期リセットおよびモジュロ最大値付き 4 ビット符号付きアップ カウンタの図



非同期リセットおよびモジュロ最大値付き 4 ビット符号付きアップ カウンタのピンの説明

I/O ピン	説明
C	クロック (立ち上がりエッジ)
CLR	非同期リセット (アクティブ High)
Q	データ出力

非同期リセットおよびモジュロ最大値付き 4 ビット符号付きアップ カウンタの VHDL コード例

```
--
-- 4-bit Signed Up Counter with Asynchronous Reset and Modulo Maximum
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity counters_8 is
    generic (MAX : integer := 16);
    port(C, CLR : in std_logic;
         Q : out integer range 0 to MAX-1);
end counters_8;

architecture archi of counters_8 is
    signal cnt : integer range 0 to MAX-1;
begin
    process (C, CLR)
    begin
        if (CLR='1') then
            cnt <= 0;
        elsif (rising_edge(C)) then
            cnt <= (cnt + 1) mod MAX ;
        end if;
    end process;

    Q <= cnt;
end archi;
```

非同期リセットおよびモジュロ最大値付き 4 ビット符号付きアップ カウンタの Verilog コード例

```
//
// 4-bit Signed Up Counter with Asynchronous Reset and Modulo Maximum
//

module v_counters_8 (C, CLR, Q);
    parameter
        MAX_SQRT = 4,
        MAX = (MAX_SQRT*MAX_SQRT);

    input  C, CLR;
    output [MAX_SQRT-1:0] Q;
    reg    [MAX_SQRT-1:0] cnt;

    always @ (posedge C or posedge CLR)
    begin
        if (CLR)
            cnt <= 0;
        else
            cnt <= (cnt + 1) %MAX;
        end

        assign Q = cnt;
    endmodule
```

アキュムレータの HDL コーディング手法

アキュムレータとカウンタは、加算および減算のオペランドが異なります。

カウンタでは、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 です。

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

アキュムレータでは、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
```

推論されるアキュムレータは、アップ、ダウン、またはアップ/ダウンになります。アップ/ダウン アキュムレータの場合、アップ モードとダウン モードで演算するデータを別にすることができます。

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

XST では、カウンタと同じ制御信号の付いたアキュムレータを推論できます。詳細は、「[カウンタの HDL コーディング手法](#)」を参照してください。

Virtex-4 および Virtex-5 デバイスのアキュムレータ

Virtex®-4 および Virtex-5 デバイスでは、アキュムレータを DSP48 リソースにインプリメントできます。XST では、DSP48 ブロックに 最大 2 段の入力レジスタを挿入できます。

アキュムレータをインプリメントできるのは、必要な DSP リソースが 1 つのみの場合のみです。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 リソースへのマクロ インプリメンテーションは、[DSP48 の使用 \(USE_DSP48\)](#) 制約またはコマンドライン オプションでデフォルト auto に設定すると制御されます。このモードでは、アキュムレータがインプリメントされる際に、デバイス上で DSP48 リソースが考慮されます。

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 DSP48 リソースが制御されます。XST では、デフォルトで DSP48 リソースをすべて使用しようとします。詳細は、「[DSP48 ブロック リソース](#)」を参照してください。

XST では、DSP48 にできるだけ多くのレジスタを含めるなど、最良のパフォーマンスを得るために最大限のマクロ 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、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。

ほかのファミリ同様、Virtex-4 および Virtex-5 の場合も、推論されたアキュムレータの詳細が HDL の合成段階でレポートされますが、アキュムレータは MAC インプリメンテーションに含まれるため、最終段階の合成レポートではレポートされません。

アキュムレータのログ ファイル

XST ログ ファイルには、認識されたアキュムレータのタイプおよびビット幅が示されます。

```
...
Synthesizing Unit <accum>.
  Related source file is accumulators_1.vhd.
  Found 4-bit up accumulator for signal <tmp>.
  Summary:
    inferred    1 Accumulator(s).
Unit <accum> synthesized.
```

```
=====
HDL Synthesis Report
```

```
Macro Statistics
# Accumulators           : 1
  4-bit up accumulator   : 1
=====
...
```

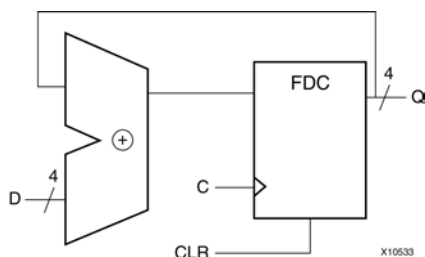
アキュムレータ関連の制約

- ・ DSP48 の使用 (USE_DSP48)
- ・ DSP 使用率 (DSP_UTILIZATION_RATIO)
- ・ キープ (KEEP)

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。アップデートは、
ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

非同期リセット付き 4 ビット符号なしアップ アキュムレータの図



非同期リセット付き 4 ビット符号なしアップ アキュムレータのピンの説明

I/O ピン	説明
C	クロック (立ち上がりエッジ)
CLR	非同期リセット (アクティブ High)
D	データ入力
Q	データ出力

非同期リセット付き 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 Asynchronous Reset  
--  
  
library ieee;  
use ieee.std_logic_1164.all;  
use ieee.std_logic_unsigned.all;  
  
entity accumulators_1 is  
    port(C, CLR : in std_logic;  
          D : in std_logic_vector(3 downto 0);  
          Q : out std_logic_vector(3 downto 0));  
end accumulators_1;  
  
architecture archi of accumulators_1 is  
    signal tmp: std_logic_vector(3 downto 0);  
begin  
  
    process (C, CLR)  
    begin  
        if (CLR='1') then  
            tmp <= "0000";  
        elsif (C'event and C='1') then  
            tmp <= tmp + D;  
        end if;  
    end process;  
  
    Q <= tmp;  
  
end archi;
```

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

```
//  
// 4-bit Unsigned Up Accumulator with Asynchronous Reset  
//  
  
module v_accumulators_1 (C, CLR, D, Q);  
  
    input C, CLR;  
    input [3:0] D;  
    output [3:0] Q;  
    reg [3:0] tmp;  
  
    always @(posedge C or posedge CLR)  
    begin  
        if (CLR)  
            tmp = 4'b0000;  
        else  
            tmp = tmp + D;  
        end  
        assign Q = tmp;  
    endmodule
```

シフトレジスタの HDL コーディング手法

通常シフトレジスタには次の制御信号およびデータ信号があり、XST で完全に認識されます。

- ・ クロック
- ・ シリアル入力
- ・ 非同期セット/リセット
- ・ 同期セット/リセット
- ・ 同期/非同期パラレル ロード
- ・ クロック イネーブル

- ・ シリアル/パラレル出力 シフトレジスタの出力モード :
 - シリアル
最後のフリップフロップのデータのみを出力
 - パラレル
1 つまたは複数のフリップフロップのデータを出力
- ・ シフト方向 : 左、右など

シフトレジスタの記述

シフトレジスタは、VHDL では次のように記述します。

- ・ 連結演算子

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

- ・ for - loop 文

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

- ・ SLL、SRL などのあらかじめ定義されたシフト演算子

詳細は、VHDL または Verilog 言語のリファレンス マニュアルを参照してください。

シフトレジスタのインプリメンテーション

シフトレジスタをインプリメントするためのハードウェア リソース

デバイス	SRL16	SRL16E	SRLC16	SRLC16E	SRLC32E
Spartan®-3	○	○	○	○	×
Spartan-3E					
Spartan-3A					
Virtex®-4	○	○	○	○	×
Virtex-5	○	○	○	○	○

SRL16 および SRLC16

SRL16 および SRLC16 には、クロック イネーブル付きの要素 (SRL16E および SRLC16E) もあります。

SLRC16x プリミティブでは、同期および非同期制御信号は使用できません。ただし、XST では、シフトレジスタの記述に非同期または同期のセット/リセット信号が 1 つ含まれると、SRL 専用のリソースが使用されるので、エリアがかなり削減できます。

SRL16 および SRLC16 の場合、次の I/O 信号では左シフトしかサポートされていません。

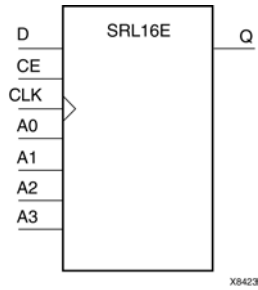
- ・ クロック
- ・ クロック イネーブル
- ・ シリアル データ入力
- ・ シリアル データ出力

たとえばシフトレジスタに同期パラレルロードまたは複数のセット/リセット信号がある場合、SRL16 はインプリメントされません。この場合、最適な結果が得られるよう、XST で特殊な内部処理が行われます。

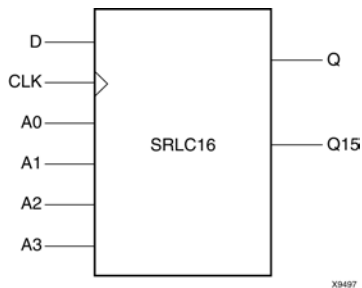
SRL16 または SRLC16 を使用したインプリメントが可能な場合は、XST のログファイルに認識されたシフトレジスタが示されます。このため、次のコード例の中には（特にパラレルロードまたはパラレル出力が付いた例）、特定のシフトレジスタがレポートされないものもあります。

詳細は、「[INIT および RLOC の指定](#)」を参照してください。

SRL16E のピン配置図



SRLC16 のピン配置図



シフトレジスタのログ ファイル

XST では、下位レベルの最適化段階でシフトレジスタが認識され、ログ ファイルにそのシフトレジスタのビット幅がレポートされます。

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

Synthesizing Unit <shift_registers_1>.
  Related source file is "shift_registers_1.vhd".
  Found 8-bit register for signal <tmp>.
  Summary:
    inferred    8 D-type flip-flop(s).
Unit <shift_registers_1> synthesized.

=====
*          Advanced HDL Synthesis          *
=====
Advanced HDL Synthesis Report
Macro Statistics
# Registers : 8
Flip-Flops : 8
=====
*          Low Level Synthesis             *
=====
Processing Unit <shift_registers_1> :
  Found 8-bit shift register for signal <tmp_7>.
Unit <shift_registers_1> processed.
=====
Final Register Report
Macro Statistics
# Shift Registers : 1
  8-bit shift register : 1
=====
```

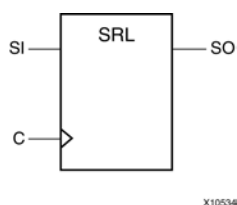
シフトレジスタ関連の制約

シフトレジスタの抽出 (SHREG_EXTRACT)

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。アップデートは、
[ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip](http://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip) からダウンロードしてください。

クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の 8 ビット シフト レジスタの図



クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の 8 ビット シフト レフト レジスタのピンの説明

I/O ピン	説明
C	クロック (立ち上がりエッジ)
SI	シリアル入力
SO	シリアル出力

クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の 8 ビット シフト レフト レジスタの VHDL コード例

```
--
-- 8-bit Shift-Left Register with Positive-Edge Clock, Serial In, and Serial Out
--

library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_1 is
    port(C, SI : in std_logic;
         SO : out std_logic);
end shift_registers_1;

architecture archi of shift_registers_1 is
    signal tmp: std_logic_vector(7 downto 0);
begin

    process (C)
    begin
        if (C'event and C='1') then
            for i in 0 to 6 loop
                tmp(i+1) <= tmp(i);
            end loop;
            tmp(0) <= SI;
        end if;
    end process;

    SO <= tmp(7);

end archi;
```

クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の 8 ビット シフト レフト レジスタの Verilog コード例

```
//
// 8-bit Shift-Left Register with Positive-Edge Clock,
// Serial In, and Serial Out
//

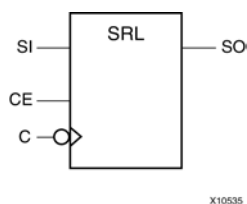
module v_shift_registers_1 (C, SI, SO);
    input C,SI;
    output SO;
    reg [7:0] tmp;

    always @(posedge C)
    begin
        tmp = {tmp[6:0], SI};
    end

    assign SO = tmp[7];

endmodule
```


クロックの立ち下がリエッジで動作するシリアル入力、シリアル出力のクロック イネーブル付き 8 ビット シフト レフト レジスタの図



クロックの立ち下がリエッジで動作するシリアル入力、シリアル出力のクロック イネーブル付き 8 ビット シフト レフト レジスタのピンの説明

I/O ピン	説明
C	クロック (立ち下がリエッジ)
SI	シリアル入力
CE	クロック イネーブル (アクティブ High)
SO	シリアル出力

クロックの立ち下がリエッジで動作するシリアル入力、シリアル出力のクロック イネーブル付き 8 ビット シフト レフト レジスタの VHDL コード例

```
--
-- 8-bit Shift-Left Register with Negative-Edge Clock, Clock Enable,
-- Serial In, and Serial Out
--

library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_2 is
    port(C, SI, CE : in std_logic;
         SO : out std_logic);
end shift_registers_2;

architecture archi of shift_registers_2 is
    signal tmp: std_logic_vector(7 downto 0);
begin

    process (C)
    begin
        if (C'event and C='0') then
            if (CE='1') then
                for i in 0 to 6 loop
                    tmp(i+1) <= tmp(i);
                end loop;
                tmp(0) <= SI;
            end if;
        end if;
    end process;

    SO <= tmp(7);

end archi;
```

クロックの立ち下がリエッジで動作するシリアル入力、シリアル出力のクロック イネーブル付き 8 ビット シフト レジスタの Verilog コード例

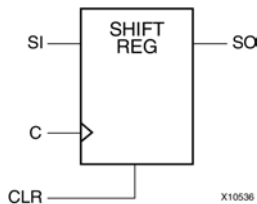
```
//
// 8-bit Shift-Left Register with Negative-Edge Clock, Clock Enable,
// Serial In, and Serial Out
//

module v_shift_registers_2 (C, CE, SI, SO);
    input C, SI, CE;
    output SO;
    reg [7:0] tmp;

    always @(negedge C)
    begin
        if (CE)
        begin
            tmp = {tmp[6:0], SI};
        end
    end

    assign SO = tmp[7];
endmodule
```

クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の非同期リセット付き 8 ビット シフト レジスタの図



クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の非同期リセット付き 8 ビット シフト レジスタのピンの説明

I/O ピン	説明
C	クロック (立ち上がりエッジ)
SI	シリアル入力
CLR	非同期リセット (アクティブ High)

クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の非同期リセット付き 8 ビット シフト レジスタの VHDL コード例

```
--
-- 8-bit Shift-Left Register with Positive-Edge Clock,
-- Asynchronous Reset, Serial In, and Serial Out
--

library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_3 is
    port(C, SI, CLR : in std_logic;
         SO : out std_logic);
end shift_registers_3;

architecture archi of shift_registers_3 is
    signal tmp: std_logic_vector(7 downto 0);
begin

    process (C, CLR)
    begin
        if (CLR='1') then
            tmp <= (others => '0');
        elsif (C'event and C='1') then
            tmp <= tmp(6 downto 0) & SI;
        end if;
    end process;

    SO <= tmp(7);

end archi;
```

クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の非同期リセット付き 8 ビット シフト レジスタの Verilog コード例

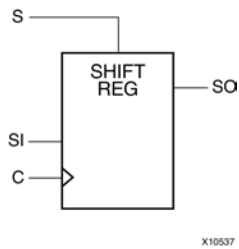
```
//
// 8-bit Shift-Left Register with Positive-Edge Clock,
// Asynchronous Reset, Serial In, and Serial Out
//

module v_shift_registers_3 (C, CLR, SI, SO);
    input C,SI,CLR;
    output SO;
    reg [7:0] tmp;

    always @(posedge C or posedge CLR)
    begin
        if (CLR)
            tmp <= 8'b00000000;
        else
            tmp <= {tmp[6:0], SI};
        end

        assign SO = tmp[7];
    endmodule
```

クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の同期セット付き 8 ビット シフト レフト レジスタの図



クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の同期セット付き 8 ビット シフト レフト レジスタのピンの説明

I/O ピン	説明
C	クロック (立ち上がりエッジ)
SI	シリアル入力
S	同期セット (アクティブ High)
SO	シリアル出力

クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の同期セット付き 8 ビット シフト レフト レジスタの VHDL コード例

```
--
-- 8-bit Shift-Left Register with Positive-Edge Clock, Synchronous Set,
-- Serial In, and Serial Out
--

library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_4 is
    port(C, SI, S : in std_logic;
         SO : out std_logic);
end shift_registers_4;

architecture archi of shift_registers_4 is
    signal tmp: std_logic_vector(7 downto 0);
begin

    process (C, S)
    begin
        if (C'event and C='1') then
            if (S='1') then
                tmp <= (others => '1');
            else
                tmp <= tmp(6 downto 0) & SI;
            end if;
        end if;
    end process;

    SO <= tmp(7);

end archi;
```

クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の同期セット付き 8 ビット シフト レフト レジスタの Verilog コード例

```
//
// 8-bit Shift-Left Register with Positive-Edge Clock, Synchronous Set,
// Serial In, and Serial Out
//

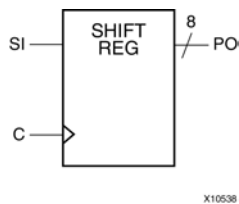
module v_shift_registers_4 (C, S, SI, SO);
    input C,SI,S;
    output SO;
    reg [7:0] tmp;

    always @(posedge C)
    begin
        if (S)
            tmp <= 8'b11111111;
        else
            tmp <= {tmp[6:0], SI};
        end

    assign SO = tmp[7];

endmodule
```

クロックの立ち上がりエッジで動作するシリアル入力、パラレル出力の 8 ビット シフト レフト レジスタの図



クロックの立ち上がりエッジで動作するシリアル入力、パラレル出力の 8 ビット シフト レフト レジスタのピンの説明

I/O ピン	説明
C	クロック (立ち上がりエッジ)
SI	シリアル入力
PO	パラレル出力

クロックの立ち上がりエッジで動作するシリアル入力、パラレル出力の 8 ビット シフト レフト レジスタの VHDL コード例

```
--
-- 8-bit Shift-Left Register with Positive-Edge Clock,
-- Serial In, and Parallel Out
--

library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_5 is
    port(C, SI : in std_logic;
         PO : out std_logic_vector(7 downto 0));
end shift_registers_5;

architecture archi of shift_registers_5 is
    signal tmp: std_logic_vector(7 downto 0);
begin

    process (C)
    begin
        if (C'event and C='1') then
            tmp <= tmp(6 downto 0) & SI;
        end if;
    end process;

    PO <= tmp;

end archi;
```

クロックの立ち上がりエッジで動作するシリアル入力、パラレル出力の 8 ビット シフト レフト レジスタの Verilog コード例

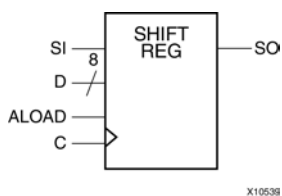
```
//
// 8-bit Shift-Left Register with Positive-Edge Clock,
// Serial In, and Parallel Out
//

module v_shift_registers_5 (C, SI, PO);
    input C, SI;
    output [7:0] PO;
    reg [7:0] tmp;

    always @(posedge C)
        tmp <= {tmp[6:0], SI};

    assign PO = tmp;
endmodule
```

クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の非同期パラレルロード付き 8 ビット シフト レフト レジスタの図



クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の非同期パラレルロード付き 8 ビット シフト レフト レジスタのピンの説明

I/O ピン	説明
C	クロック (立ち上がりエッジ)
SI	シリアル入力
ALOAD	非同期パラレルロード (アクティブ High)
D	データ入力
SO	シリアル出力

クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の非同期パラレルロード付き 8 ビット シフト レフト レジスタの VHDL コード例

```
--
-- 8-bit Shift-Left Register with Positive-Edge Clock,
-- Asynchronous Parallel Load, Serial In, and Serial Out
--

library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_6 is
    port(C, SI, ALOAD : in std_logic;
         D : in std_logic_vector(7 downto 0);
         SO : out std_logic);
end shift_registers_6;

architecture archi of shift_registers_6 is
    signal tmp: std_logic_vector(7 downto 0);
begin

    process (C, ALOAD, D)
    begin
        if (ALOAD='1') then
            tmp <= D;
        elsif (C'event and C='1') then
            tmp <= tmp(6 downto 0) & SI;
        end if;
    end process;

    SO <= tmp(7);

end archi;
```

クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の非同期パラレル ロード付き 8 ビット シフト レフト レジスタの Verilog コード例

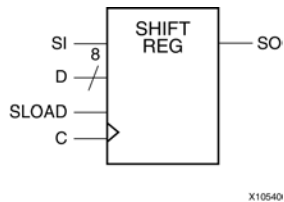
```
//
// 8-bit Shift-Left Register with Positive-Edge Clock,
// Asynchronous Parallel Load, Serial In, and Serial Out
//

module v_shift_registers_6 (C, ALOAD, SI, D, SO);
    input C,SI,ALOAD;
    input [7:0] D;
    output SO;
    reg [7:0] tmp;

    always @(posedge C or posedge ALOAD)
    begin
        if (ALOAD)
            tmp <= D;
        else
            tmp <= {tmp[6:0], SI};
        end

    assign SO = tmp[7];
endmodule
```

クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の同期パラレル ロード付き 8 ビット シフト レフト レジスタの図



クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の同期パラレル ロード付き 8 ビット シフト レフト レジスタのピンの説明

I/O ピン	説明
C	クロック (立ち上がりエッジ)
SI	シリアル入力
SLOAD	同期パラレル ロード (アクティブ High)
D	データ入力
SO	シリアル出力

クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の同期パラレル ロード付き 8 ビット シフト レフト レジスタの VHDL コード例

```
--
-- 8-bit Shift-Left Register with Positive-Edge Clock,
-- Synchronous Parallel Load, Serial In, and Serial Out
--

library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_7 is
    port(C, SI, SLOAD : in std_logic;
         D : in std_logic_vector(7 downto 0);
         SO : out std_logic);
end shift_registers_7;

architecture archi of shift_registers_7 is
    signal tmp: std_logic_vector(7 downto 0);
begin

    process (C)
    begin
        if (C'event and C='1') then
            if (SLOAD='1') then
                tmp <= D;
            else
                tmp <= tmp(6 downto 0) & SI;
            end if;
        end if;
    end process;

    SO <= tmp(7);

end archi;
```

クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の同期パラレル ロード付き 8 ビット シフト レフト レジスタの Verilog コード例

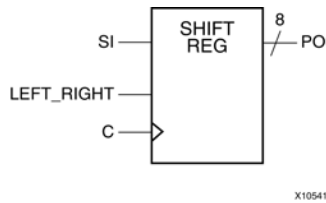
```
//
// 8-bit Shift-Left Register with Positive-Edge Clock,
// Synchronous Parallel Load, Serial In, and Serial Out
//

module v_shift_registers_7 (C, SLOAD, SI, D, SO);
    input C,SI,SLOAD;
    input [7:0] D;
    output SO;
    reg [7:0] tmp;

    always @(posedge C)
    begin
        if (SLOAD)
            tmp <= D;
        else
            tmp <= {tmp[6:0], SI};
        end

        assign SO = tmp[7];
    endmodule
```

クロックの立ち上がりエッジで動作するシリアル入力、パラレル出力の 8 ビット シフト レフト/シフト ライト レジスタの図



クロックの立ち上がりエッジで動作するシリアル入力、パラレル出力の 8 ビット シフト レフト/シフト ライト レジスタのピンの説明

I/O ピン	説明
C	クロック (立ち上がりエッジ)
SI	シリアル入力
LEFT_RIGHT	左右のシフト方向セクタ
PO	パラレル出力

クロックの立ち上がりエッジで動作するシリアル入力、パラレル出力の 8 ビット シフト レフト/シフト ライト レジスタの VHDL コード例

```
--
-- 8-bit Shift-Left/Shift-Right Register with Positive-Edge Clock,
-- Serial In, and Parallel Out
--

library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_8 is
    port(C, SI, LEFT_RIGHT : in std_logic;
         PO : out std_logic_vector(7 downto 0));
end shift_registers_8;

architecture archi of shift_registers_8 is
    signal tmp: std_logic_vector(7 downto 0);
begin

    process (C)
    begin
        if (C'event and C='1') then
            if (LEFT_RIGHT='0') then
                tmp <= tmp(6 downto 0) & SI;
            else
                tmp <= SI & tmp(7 downto 1);
            end if;
        end if;
    end process;

    PO <= tmp;

end archi;
```

クロックの立ち上がりエッジで動作するシリアル入力、パラレル出力の 8 ビット シフト レフト/シフト ライト レジスタの Verilog コード例

```
//
// 8-bit Shift-Left/Shift-Right Register with Positive-Edge Clock,
// Serial In, and Parallel Out
//

module v_shift_registers_8 (C, SI, LEFT_RIGHT, PO);
    input C,SI,LEFT_RIGHT;
    output [7:0] PO;
    reg [7:0] tmp;

    always @(posedge C)
    begin
        if (LEFT_RIGHT==1'b0)
            tmp <= {tmp[6:0], SI};
        else
            tmp <= {SI, tmp[7:1]};
        end

    assign PO = tmp;
endmodule
```

ダイナミック シフト レジスタの HDL コーディング手法

XST では、ダイナミック シフト レジスタを推論できます。ダイナミック シフト レジスタが認識されると、その特性が XST マクロ生成機能に渡され、のプリミティブを使用して最適なインプリメンテーションが行われます。

デバイス	SRL16	SRL16E	SRLC16	SRLC16E	SRLC32E
Spartan®-3	○	○	○	○	×
Spartan-3E					
Spartan-3A					
Virtex®-4	○	○	○	○	×
Virtex-5	○	○	○	○	○

ダイナミック シフト レジスタのログ ファイル

ダイナミック シフト レジスタは、アドバンス HDL 合成の段階で認識されます。XST ログ ファイルには、認識されたダイナミック シフト レジスタのビット幅が示されます。

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

Synthesizing Unit <dynamic_shift_registers_1>.
  Related source file is "dynamic_shift_registers_1.vhd".
  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 <dynamic_shift_registers_1> synthesized.

=====
*                      Advanced HDL Synthesis              *
=====
...
Synthesizing (advanced) Unit <dynamic_shift_registers_1>.
  Found 16-bit dynamic shift register for signal <Q>.
```

```
Unit <dynamic_shift_registers_1> synthesized (advanced).
```

```
=====
HDL Synthesis Report
```

```
Macro Statistics
# Shift Registers          : 1
  16-bit dynamic shift register : 1
```

```
=====
...
```

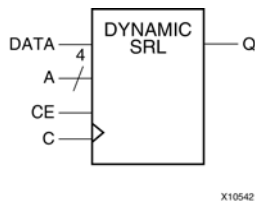
ダイナミック シフト レジスタ関連の制約

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。アップデートは、
ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の 16 ビット ダイナミック シフト レフト レジスタの図



次は、ダイナミック シフトレジスタのピン定義を示します。レジスタは、次のようにできます。

- ・ シリアルまたはパラレル
- ・ レフトまたはライト
- ・ 同期または非同期リセット付き
- ・ 深さは最大 16 ビットまで

クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の 16 ビット ダイナミック シフト レフト レジスタのピンの説明

I/O ピン	説明
C	クロック (立ち上がりエッジ)
SI	シリアル入力
AClr	非同期リセット
SClr	同期リセット
SLoad	同期パラレル ロード
Data	パラレル データ入力ポート
ClkEn	クロック イネーブル
LeftRight	シフト方向選択
SerialInRight	双方向シフトレジスタ用シリアル入力ライト
PSO	シリアル/パラレル出力

クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の 16 ビット ダイナミック シフト レフト レジスタの VHDL コード例

```
--
-- 16-bit dynamic shift register.
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity dynamic_shift_registers_1 is
    port(CLK : in std_logic;
         DATA : in std_logic;
         CE : in std_logic;
         A : in std_logic_vector(3 downto 0);
         Q : out std_logic);
end dynamic_shift_registers_1;

architecture rtl of dynamic_shift_registers_1 is
    constant DEPTH_WIDTH : integer := 16;

    type SRL_ARRAY is array (0 to DEPTH_WIDTH-1) of std_logic;
    -- The type SRL_ARRAY can be array
    -- (0 to DEPTH_WIDTH-1) of
    -- std_logic_vector(BUS_WIDTH downto 0)
    -- or array (DEPTH_WIDTH-1 downto 0) of
    -- std_logic_vector(BUS_WIDTH downto 0)
    -- (the subtype is forward (see below))
    signal SRL_SIG : SRL_ARRAY;

begin
    PROC_SRL16 : process (CLK)
    begin
        if (CLK'event and CLK = '1') then
            if (CE = '1') then
                SRL_SIG <= DATA & SRL_SIG(0 to DEPTH_WIDTH-2);
            end if;
        end if;
    end process;

    Q <= SRL_SIG(conv_integer(A));

end rtl;
```

クロックの立ち上がりエッジで動作するシリアル入力、シリアル出力の 16 ビット ダイナミック シフト レフト レジスタの Verilog コード例

```
//
// 16-bit dynamic shift register.
//

module v_dynamic_shift_registers_1 (Q,CE,CLK,D,A);
    input CLK, D, CE;
    input [3:0] A;
    output Q;
    reg [15:0] data;

    assign Q = data[A];

    always @(posedge CLK)
    begin
        if (CE == 1'b1)
            data <= {data[14:0], D};
        end
    endmodule
```

マルチプレクサの HDL コーディング手法

XST では、if-then-else 文や case 文などを使用した、さまざまなマルチプレクサ (MUX) の記述方法がサポートされています。

たとえば、case 文を使用してマルチプレクサを記述する場合に、セレクトのすべての値を指定しないと、マルチプレクサではなくラッチが推論されてしまいます。MUX を記述する際には、セレクトの値に「ドントケア」も使用できます。

XST では、マクロ推論段階で MUX を推論するかどうかが決まります。MUX に同じ入力がある場合、MUX は推論されません。MUX を強制的に推論させる場合は、MUX_EXTRACT 制約を使用します。

Verilog の case 文は、次のように指定できます。

- ・ **full** または **not full**
- ・ **parallel** または **not parallel**

このフルとパラレルは、次の場合に指定します。

- ・ **full** : 可能なすべての分岐が指定されている場合
- ・ **parallel** : 同時に実行可能な分岐が含まれていない場合

マルチプレクサのフルおよびパラレルの case 文のコード例

```
module full (sel, i1, i2, i3, i4, o1);
input [1:0] sel;
input [1:0] i1, i2, i3, i4;
output [1:0] o1;

    reg [1:0] o1;

always @(sel or i1 or i2 or i3 or i4)
begin
    case (sel)
        2'b00: o1 = i1;
        2'b01: o1 = i2;
        2'b10: o1 = i3;
        2'b11: o1 = i4;
    endcase
end
endmodule
```

マルチプレクサのフルではないがパラレルの case 文のコード例

```
module notfull (sel, i1, i2, i3, o1);
input [1:0] sel;
input [1:0] i1, i2, i3;
output [1:0] o1;

    reg [1:0] o1;

always @(sel or i1 or i2 or i3)
begin
    case (sel)
        2'b00: o1 = i1;
        2'b01: o1 = i2;
        2'b10: o1 = i3;
    endcase
end
endmodule
```

マルチプレクサのフルでもパラレルでもない case 文のコード例

```
module notfull_notparallel (sel1, sel2, i1, i2, o1);
    input [1:0] sel1, sel2;
    input [1:0] i1, i2;
    output [1:0] o1;

    reg [1:0] o1;

    always @(sel1 or sel2)
    begin
        case (2'b00)
            sel1: o1 = i1;
            sel2: o1 = i2;
        endcase
    end
endmodule
```

XST は case 文の特性を自動的に判断し、その case 文のビヘイビアを正確にインプリメントするロジックをマルチプレクサ、プライオリティ エンコーダ、およびラッチを使用して生成します。

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

[Case Implementation Style] プロパティ (-vlgcase オプション) を使用すると、case 文の特性を強制的に指定できます。詳細は、「[デザイン制約](#)」を参照してください。このパラメータに使用できる値は、none、full、parallel、full-parallel のいずれかです。

- ・ [None] (デフォルト) に指定した場合、case 文のビヘイビアがそのままインプリメントされます。
- ・ [Full] に設定すると case 文は完全と見なされ、ラッチは作成されません。
- ・ [Parallel] に設定すると、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] に設定すると、case 文は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。

次の Verilog の case 文のリソースの表は、[Case Implementation Style] の 4 つのスタイルでマルチプレクサの case 文の例を合成する場合に、使用されるリソースを示しています。ここで「リソース」とは機能を意味します。たとえば、フルでもパラレルでもない case 文で [Case Implementation Style] を [None] に設定すると、機能の点でプライオリティ エンコーダとラッチがインプリメントされますが、マクロ認識の段階でプライオリティ エンコーダが推論されるとは限りません。

マルチプレクサの Verilog の case 文のリソース

パラメータ値	case 文のインプリメンテーション		
	Full	Not Full	フルでもパラレルでもない
なし	MUX	ラッチ	プライオリティ エンコーダ + ラッチ
parallel	MUX	ラッチ	ラッチ
full	MUX	MUX	プライオリティ エンコーダ
full-parallel	MUX	MUX	MUX

[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。

マルチプレクサのログ ファイル

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。

```
...
Synthesizing Unit <mux>.
  Related source file is multiplexers_1.vhd.
  Found 1-bit 4-to-1 multiplexer for signal <o>.
  Summary:
    inferred    1 Multiplexer(s).
  Unit <mux> synthesized.

=====
HDL Synthesis Report

Macro Statistics
# Multiplexers           : 1
  1-bit 4-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 デバイスによって異なります。次のコード例は、4:1 マルチプレクサに制限されます。これらはターゲットが LUT ベースのデバイス ファミリの場合でのみ上記のようにレポートされます。Virtex®-5 デバイスの場合、マルチプレクサは 8:1 以上のサイズでのみ明示的に推論されます。

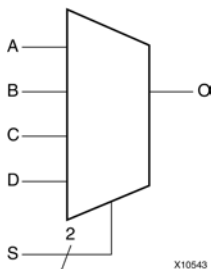
マルチプレクサ関連の制約

- ・ MUX の抽出 (MUX_EXTRACT)
- ・ MUX スタイル (MUX_STYLE)
- ・ 列挙型エンコード手法 (ENUM_ENCODING)

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。アップデートは、
ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

if 文を使用した 4:1 の 1 ビット MUX の図



if 文を使用した 4:1 の 1 ビット MUX のピンの説明

I/O ピン	説明
a、b、c、d	データ入力
s	MUX セレクタ
o	データ出力

if 文を使用した 4:1 の 1 ビット MUX の VHDL コード例

```
--
-- 4-to-1 1-bit MUX using an If statement.
--

library ieee;
use ieee.std_logic_1164.all;

entity multiplexers_1 is
    port (a, b, c, d : in std_logic;
          s : in std_logic_vector (1 downto 0);
          o : out std_logic);
end multiplexers_1;

architecture archi of multiplexers_1 is
begin
    process (a, b, c, d, s)
    begin
        if (s = "00") then o <= a;
        elsif (s = "01") then o <= b;
        elsif (s = "10") then o <= c;
        else o <= d;
        end if;
    end process;
end archi;
```

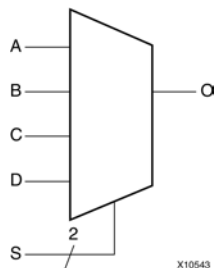
if 文を使用した 4:1 の 1 ビット MUX の Verilog コード例

```
//
// 4-to-1 1-bit MUX using an If statement.
//

module v_multiplexers_1 (a, b, c, d, s, o);
    input a,b,c,d;
    input [1:0] s;
    output o;
    reg o;

    always @(a or b or c or d or s)
    begin
        if (s == 2'b00) o = a;
        else if (s == 2'b01) o = b;
        else if (s == 2'b10) o = c;
        else o = d;
    end
endmodule
```

case 文を使用した 4:1 の 1 ビット MUX の図



case 文を使用した 4:1 の 1 ビット MUX のピンの説明

I/O ピン	説明
a、b、c、d	データ入力
s	MUX セレクタ
o	データ出力

case 文を使用した 4:1 の 1 ビット MUX の VHDL コード例

```
--
-- 4-to-1 1-bit MUX using a Case statement.
--

library ieee;
use ieee.std_logic_1164.all;

entity multiplexers_2 is
    port (a, b, c, d : in std_logic;
          s : in std_logic_vector (1 downto 0);
          o : out std_logic);
end multiplexers_2;

architecture archi of multiplexers_2 is
begin
    process (a, b, c, d, s)
    begin
        case s is
            when "00" => o <= a;
            when "01" => o <= b;
            when "10" => o <= c;
            when others => o <= d;
        end case;
    end process;
end archi;
```

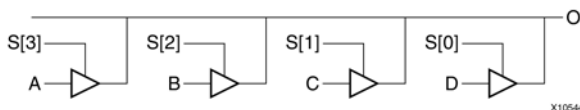
case 文を使用した 4:1 の 1 ビット MUX の Verilog コード例

```
//
// 4-to-1 1-bit MUX using a Case statement.
//

module v_multiplexers_2 (a, b, c, d, s, o);
    input a,b,c,d;
    input [1:0] s;
    output o;
    reg o;

    always @(a or b or c or d or s)
    begin
        case (s)
            2'b00 : o = a;
            2'b01 : o = b;
            2'b10 : o = c;
            default : o = d;
        endcase
    end
endmodule
```

トリステート バッファを使用した 4:1 の 1 ビット MUX の図



トリステート バッファを使用した 4:1 の 1 ビット MUX のピンの説明

I/O ピン	説明
a、b、c、d	データ入力
s	MUX セレクタ
o	データ出力

トリステート バッファを使用した 4:1 の 1 ビット MUX の VHDL コード例

```
--
-- 4-to-1 1-bit MUX using tristate buffers.
--

library ieee;
use ieee.std_logic_1164.all;

entity multiplexers_3 is
    port (a, b, c, d : in std_logic;
          s : in std_logic_vector (3 downto 0);
          o : out std_logic);
end multiplexers_3;

architecture archi of multiplexers_3 is
begin
    o <= a when (s(0)='0') else 'Z';
    o <= b when (s(1)='0') else 'Z';
    o <= c when (s(2)='0') else 'Z';
    o <= d when (s(3)='0') else 'Z';
end archi;
```

トリステート バッファを使用した 4:1 の 1 ビット MUX の Verilog コード例

```
//
// 4-to-1 1-bit MUX using tristate buffers.
//

module v_multiplexers_3 (a, b, c, d, s, o);
    input a,b,c,d;
    input [3:0] s;
    output o;

    assign o = s[3] ? a :1'bz;
    assign o = s[2] ? b :1'bz;
    assign o = s[1] ? c :1'bz;
    assign o = s[0] ? d :1'bz;
endmodule
```

次のコード例では、if - elseif 文の終わりに else 文を使用しなかった場合に XST でラッチがどのように推論されるかを示しています。else 文がないので、XST では s=11 の場合に o に以前の値が保持され、メモリ エLEMENTが必要となると認識されます。XST では、次のような警告メッセージが表示されます。

```
WARNING:Xst:737 - Found 1-bit latch for signal <o1>.
INFO:Xst - HDL ADVISOR - Logic functions respectively driving the data
and gate enable inputs of this latch share common terms. This situation
will potentially lead to setup/hold violations and, as a result, to
simulation problems. This situation may come from an incomplete case statement
(all selector values are not covered). You should carefully review if it was
in your intentions to describe such a latch.
```

このメッセージは、ラッチのデータ入力およびゲート イネーブル入力をそれぞれ駆動するロジック ファンクションが同じ項を共有していて、このためにセットアップ/ホールド違反になることがあることを示しています。このようなラッチを記述する意図がある場合を除き、else 文を追加してください。

注意： else 文を記述しないと、シミュレーションでエラーになることがあります。

ラッチ推論になる else 文が記述されていない VHDL コード例

```
--
-- 3-to-1 1-bit MUX with a 1-bit latch.
--

library ieee;
use ieee.std_logic_1164.all;

entity multiplexers_4 is
    port (a, b, c: in std_logic;
          s : in std_logic_vector (1 downto 0);
          o : out std_logic);
end multiplexers_4;

architecture archi of multiplexers_4 is
begin
    process (a, b, c, s)
    begin
        if (s = "00") then o <= a;
        elsif (s = "01") then o <= b;
        elsif (s = "10") then o <= c;
        end if;
    end process;
end archi;
```

ラッチ推論になる else 文が記述されていない Verilog コード例

```
//
// 3-to-1 1-bit MUX with a 1-bit latch.
//
module v_multiplexers_4 (a, b, c, s, o);
    input a,b,c;
    input [1:0] s;
    output o;
    reg o;

    always @(a or b or c or s)
    begin
        if (s == 2'b00) o = a;
        else if (s == 2'b01) o = b;
        else if (s == 2'b10) o = c;
    end
endmodule
```

デコーダの HDL コーディング手法

デコーダは、各入力値に対して 1 つのワンホット (またはワンコールド) 値が指定されているマルチプレクサです。詳細は、「[マルチプレクサの HDL コーディング手法](#)」を参照してください。

デコーダのログ ファイル

XST ログ ファイルには、認識されたデコーダのタイプおよびビット幅が示されます。

```
Synthesizing Unit <dec>.
    Related source file is decoders_1.vhd.
    Found 1-of-8 decoder for signal <res>.
    Summary:
        inferred    1 Decoder(s).
    Unit <dec> synthesized.
=====
HDL Synthesis Report

Macro Statistics
# Decoders                      : 1
```

```

1-of-8 decoder                                : 1
=====
...

```

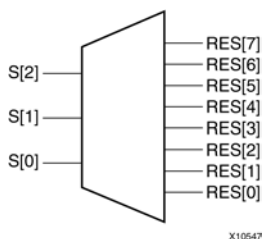
デコーダ関連の制約

デコーダの抽出 (DECODER_EXTRACT)

デコーダのコード例

コード例は、本書が作成された時点のものです。アップデートは、
[ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip](http://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip) からダウンロードしてください。

1:8 のデコーダ (ワンホット) の図



1:8 のデコーダ (ワンホット) のピンの説明

I/O ピン	説明
s	セレクト
res	データ出力

1:8 のデコーダ (ワンホット) の VHDL コード例

```

--
-- 1-of-8 decoder (One-Hot)
--

library ieee;
use ieee.std_logic_1164.all;

entity decoders_1 is
    port (sel: in std_logic_vector (2 downto 0);
          res: out std_logic_vector (7 downto 0));
end decoders_1;

architecture archi of decoders_1 is
begin
    res <= "00000001" when sel = "000" else
           "00000010" when sel = "001" else
           "00000100" when sel = "010" else
           "00001000" when sel = "011" else
           "00010000" when sel = "100" else
           "00100000" when sel = "101" else
           "01000000" when sel = "110" else
           "10000000";
end archi;

```

1:8 のデコーダ (ワンホット) の Verilog コード例

```
//
// 1-of-8 decoder (One-Hot)
//

module v_decoders_1 (sel, res);
    input [2:0] sel;
    output [7:0] res;
    reg [7:0] res;

    always @(sel or res)
    begin
        case (sel)
            3'b000 : res = 8'b00000001;
            3'b001 : res = 8'b00000010;
            3'b010 : res = 8'b00000100;
            3'b011 : res = 8'b00001000;
            3'b100 : res = 8'b00010000;
            3'b101 : res = 8'b00100000;
            3'b110 : res = 8'b01000000;
            default : res = 8'b10000000;
        endcase
    end
endmodule
```

1:8 のデコーダ (ワンコールド) のピンの説明

I/O ピン	説明
s	セレクト
res	データ出力

1:8 のデコーダ (ワンコールド) の VHDL コード例

```
--
-- 1-of-8 decoder (One-Cold)
--

library ieee;
use ieee.std_logic_1164.all;

entity decoders_2 is
    port (sel: in std_logic_vector (2 downto 0);
          res: out std_logic_vector (7 downto 0));
end decoders_2;

architecture archi of decoders_2 is
begin
    res <= "11111110" when sel = "000" else
           "11111101" when sel = "001" else
           "11111011" when sel = "010" else
           "11110111" when sel = "011" else
           "11101111" when sel = "100" else
           "11011111" when sel = "101" else
           "10111111" when sel = "110" else
           "01111111";
end archi;
```

1:8 のデコーダ (ワンコールド) の Verilog コード例

```
//
// 1-of-8 decoder (One-Cold)
//

module v_decoders_2 (sel, res);
    input [2:0] sel;
    output [7:0] res;
    reg [7:0] res;

    always @(sel)
    begin
        case (sel)
            3'b000 : res = 8'b11111110;
            3'b001 : res = 8'b11111101;
            3'b010 : res = 8'b11111011;
            3'b011 : res = 8'b11110111;
            3'b100 : res = 8'b11101111;
            3'b101 : res = 8'b11011111;
            3'b110 : res = 8'b10111111;
            default : res = 8'b01111111;
        endcase
    end
endmodule
```

選択されない出力を持つデコーダのピンの説明

I/O ピン	説明
s	セレクト
res	データ出力

デコーダが推論されない (デコーダ出力が未使用の) VHDL コード例

```
--
-- No Decoder Inference (unused decoder output)
--

library ieee;
use ieee.std_logic_1164.all;

entity decoders_3 is
    port (sel: in std_logic_vector (2 downto 0);
          res: out std_logic_vector (7 downto 0));
end decoders_3;

architecture archi of decoders_3 is
begin
    res <= "00000001" when sel = "000" else
        -- unused decoder output
        "XXXXXXXX" when sel = "001" else
        "00000100" when sel = "010" else
        "00001000" when sel = "011" else
        "00010000" when sel = "100" else
        "00100000" when sel = "101" else
        "01000000" when sel = "110" else
        "10000000";
end archi;
```

デコーダが推論されない (デコーダ出力が未使用の) Verilog コード例

```
//  
// No Decoder Inference (unused decoder output)  
//  
module v_decoders_3 (sel, res);  
    input [2:0] sel;  
    output [7:0] res;  
    reg [7:0] res;  
  
    always @(sel)  
    begin  
        case (sel)  
            3'b000 : res = 8'b00000001;  
            // unused decoder output  
            3'b001 : res = 8'bxxxxxxxx;  
            3'b010 : res = 8'b00000100;  
            3'b011 : res = 8'b00001000;  
            3'b100 : res = 8'b00010000;  
            3'b101 : res = 8'b00100000;  
            3'b110 : res = 8'b01000000;  
            default : res = 8'b10000000;  
        endcase  
    end  
endmodule
```

デコーダが推論されない (一部のセレクト値が未使用の) VHDL コード例

```
--  
-- No Decoder Inference (some selector values are unused)  
--  
  
library ieee;  
use ieee.std_logic_1164.all;  
  
entity decoders_4 is  
    port (sel: in std_logic_vector (2 downto 0);  
          res: out std_logic_vector (7 downto 0));  
end decoders_4;  
  
architecture archi of decoders_4 is  
begin  
    res <= "00000001" when sel = "000" else  
           "00000010" when sel = "001" else  
           "00000100" when sel = "010" else  
           "00001000" when sel = "011" else  
           "00010000" when sel = "100" else  
           "00100000" when sel = "101" else  
           -- 110 and 111 selector values are unused  
           "XXXXXXXX";  
end archi;
```


デコーダが推論されない (一部のセレクト値が未使用の) Verilog コード例

```
//
// No Decoder Inference (some selector values are unused)
//

module v_decoders_4 (sel, res);
    input [2:0] sel;
    output [7:0] res;
    reg [7:0] res;

    always @(sel or res)
    begin
        case (sel)
            3'b000 : res = 8'b00000001;
            3'b001 : res = 8'b00000010;
            3'b010 : res = 8'b00000100;
            3'b011 : res = 8'b00001000;
            3'b100 : res = 8'b00010000;
            3'b101 : res = 8'b00100000;
            // 110 and 111 selector values are unused
            default : res = 8'bxxxxxxxx;
        endcase
    end
endmodule
```

プライオリティ エンコーダの HDL コーディング手法

XST ではプライオリティ エンコーダを認識できますが、ほとんどの場合プライオリティ エンコーダが推論されることはありません。プライオリティ エンコーダを強制的に推論させるには、[プライオリティ エンコーダの抽出 \(PRIORITY_EXTRACT\)](#) 制約を force オプションで使します。

[プライオリティ エンコーダの抽出 \(PRIORITY_EXTRACT\)](#) 制約は、信号ごとに使用することをお勧めします。こうしないと、[プライオリティ エンコーダの抽出 \(PRIORITY_EXTRACT\)](#) の使用により、最適な結果とならないことがあります。

プライオリティ エンコーダのログ ファイル

XST ログ ファイルには、認識されたプライオリティ エンコーダのタイプおよびビット幅が示されます。

```
...
Synthesizing Unit <priority>.
    Related source file is priority_encoders_1.vhd.
    Found 3-bit 1-of-9 priority encoder for signal <code>.
    Summary:
        inferred    3 Priority encoder(s).
    Unit <priority> synthesized.
```

```
=====
HDL Synthesis Report
```

```
Macro Statistics
# Priority Encoders          : 1
  3-bit 1-of-9 priority encoder : 1
=====
...
```

プライオリティ エンコーダ関連の制約

[プライオリティ エンコーダの抽出 \(PRIORITY_EXTRACT\)](#)

プライオリティ エンコーダのコード例

コード例は、本書が作成された時点のものです。アップデートは、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

1:9 の 3 ビットのプライオリティ エンコーダのコード例

この例では、XST でプライオリティ エンコーダが推論される場合もあります。値を force にして [プライオリティ エンコーダの抽出 \(PRIORITY_EXTRACT\)](#) 制約を使用すると、プライオリティ エンコーダが強制的に推論されます。

1:9 の 3 ビットのプライオリティ エンコーダのピンの説明

I/O ピン	説明
sel	セレクト
code	エンコードされた出力バス

1:9 の 3 ビットのプライオリティ エンコーダの VHDL コード例

```
--
-- 3-Bit 1-of-9 Priority Encoder
--

library ieee;
use ieee.std_logic_1164.all;

entity priority_encoder_1 is
    port ( sel : in std_logic_vector (7 downto 0);
          code :out std_logic_vector (2 downto 0));

    attribute priority_extract: string;
    attribute priority_extract of priority_encoder_1: entity is "force";
end priority_encoder_1;

architecture archi of priority_encoder_1 is
begin

    code <= "000" when sel(0) = '1' else
            "001" when sel(1) = '1' else
            "010" when sel(2) = '1' else
            "011" when sel(3) = '1' else
            "100" when sel(4) = '1' else
            "101" when sel(5) = '1' else
            "110" when sel(6) = '1' else
            "111" when sel(7) = '1' else
            "---";

end archi;
```

1:9 の 3 ビットのプライオリティ エンコーダの Verilog コード例

```
//  
// 3-Bit 1-of-9 Priority Encoder  
//  
  
(* priority_extract="force" *)  
module v_priority_encoder_1 (sel, code);  
    input  [7:0] sel;  
    output [2:0] code;  
    reg      [2:0] code;  
  
    always @(sel)  
    begin  
        if      (sel[0]) code = 3'b000;  
        else if (sel[1]) code = 3'b001;  
        else if (sel[2]) code = 3'b010;  
        else if (sel[3]) code = 3'b011;  
        else if (sel[4]) code = 3'b100;  
        else if (sel[5]) code = 3'b101;  
        else if (sel[6]) code = 3'b110;  
        else if (sel[7]) code = 3'b111;  
        else            code = 3'bxxx;  
    end  
  
endmodule
```

論理シフトの HDL コーディング手法

ザイリンクスでは、論理シフトを次のような 2 つの入力と 1 つの出力を持つ組み合わせ回路であると定義しています。

- ・ シフトされるデータ入力
- ・ シフト段数を決定する 2 進値のセレクタ
- ・ シフト操作の結果を示す出力

これらの I/O はすべて必須で、1 つでも欠けると論理シフトは推論されません。

Hardware Description Language (HDL) コードを記述する際には、次に注意してください。

- ・ 論理、四則演算、および回転シフト操作のみを使用します。空の位置に別の信号からの値を入力するシフト操作は認識されません。
- ・ VHDL では、あらかじめ定義されたシフト (SLL、SRL、ROL など) または連結演算のみを使用します。あらかじめ定義されたシフト操作については、IEEE 規格の VHDL リファレンス マニュアルを参照してください。
- ・ 1 つのタイプのシフト操作のみを使用します。
- ・ シフト操作の n 値は、セレクタの次の 2 進値に対して 1 だけ増分または減分させる必要があります。
- ・ n には正の値のみを使用できます。
- ・ セレクタのすべての値を記述する必要があります。

論理シフタのログ ファイル

XST ログ ファイルには、認識された論理シフタのタイプおよびビット幅が示されます。

```
...
Synthesizing Unit <lshift>.
  Related source file is Logical_Shifters_1.vhd.
  Found 8-bit shifter logical left for signal <so>.
  Summary:
    inferred    1 Combinational logic shifter(s).
  Unit <lshift> synthesized.
...
=====
HDL Synthesis Report

Macro Statistics
# Logic shifters           : 1
  8-bit shifter logical left : 1
=====
...
```

論理シフタ関連の制約

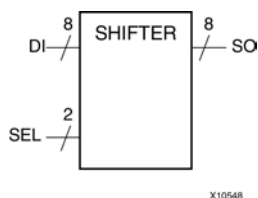
論理シフタの抽出 (SHIFT_EXTRACT)

論理シフタのコード例

コード例は、本書が作成された時点のものです。アップデートは、
ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

論理シフタ マクロを明示的に推論するために必要な XST 用の最小サイズは、ターゲット デバイスによって異なります。次のコード例は、Virtex®-4 デバイスのような LUT4 ベースのデバイス ファミリーで有効です。Virtex-5 デバイスの場合、論理シフトはセクタのサイズが少なくとも 3 の場合にのみ明示的に推論されます。

論路シフタ 1 の図



論路シフタ 1 のピンの説明

I/O ピン	説明
DI	データ入力
SEL	シフト段数セクタ
SO	データ出力

論理シフタ 1 の VHDL コード例

```
--
-- Following is the VHDL code for a logical shifter.
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity logical_shifters_1 is
    port(DI : in unsigned(7 downto 0);
         SEL : in unsigned(1 downto 0);
         SO : out unsigned(7 downto 0));
end logical_shifters_1;

architecture archi of logical_shifters_1 is
begin
    with SEL select
        SO <= DI when "00",
        DI sll 1 when "01",
        DI sll 2 when "10",
        DI sll 3 when others;
end archi;
```

論理シフタ 1 の Verilog コード例

```
//
// Following is the Verilog code for a logical shifter.
//

module v_logical_shifters_1 (DI, SEL, SO);
    input [7:0] DI;
    input [1:0] SEL;
    output [7:0] SO;
    reg [7:0] SO;

    always @(DI or SEL)
    begin
        case (SEL)
            2'b00 : SO = DI;
            2'b01 : SO = DI << 1;
            2'b10 : SO = DI << 2;
            default : SO = DI << 3;
        endcase
    end
endmodule
```

論路シフタ 2 のピンの説明

I/O ピン	説明
DI	データ入力
SEL	シフト段数セクタ
SO	データ出力

論理シフタ 2 の VHDL コード例

この例では、セクタのすべての値が定義されていないため、論理シフタ 2 の論理シフタは推論されません。

```
--
-- XST does not infer a logical shifter for this example,
-- as not all of the selector values are presented.
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity logical_shifters_2 is
    port(DI : in unsigned(7 downto 0);
          SEL : in unsigned(1 downto 0);
          SO : out unsigned(7 downto 0));
end logical_shifters_2;

architecture archi of logical_shifters_2 is
begin
    with SEL select
        SO <= DI when "00",
        DI sll 1 when "01",
        DI sll 2 when others;
end archi;
```

論理シフタ 2 の Verilog コード例

```
//
// XST does not infer a logical shifter for this example,
// as not all of the selector values are presented.
//

module v_logical_shifters_2 (DI, SEL, SO);
    input [7:0] DI;
    input [1:0] SEL;
    output [7:0] SO;
    reg [7:0] SO;

    always @(DI or SEL)
    begin
        case (SEL)
            2'b00 : SO = DI;
            2'b01 : SO = DI << 1;
            default : SO = DI << 2;
        endcase
    end
endmodule
```

論路シフタ 3 のピンの説明

I/O ピン	説明
DI	データ入力
SEL	シフト段数セクタ
SO	データ出力

論理シフタ 3 の VHDL コード例

次の例では、セクタの値が次の 2 進値で 1 増分していないので、論理シフタは推論されません。

```
--
-- XST does not infer a logical shifter for this example,
-- as the value is not incremented by 1 for each consequent
-- binary value of the selector.
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity logical_shifters_3 is
    port(DI : in unsigned(7 downto 0);
         SEL : in unsigned(1 downto 0);
         SO : out unsigned(7 downto 0));
end logical_shifters_3;

architecture archi of logical_shifters_3 is
begin
    with SEL select
        SO <= DI when "00",
        DI sll 1 when "01",
        DI sll 3 when "10",
        DI sll 2 when others;
end archi;
```

論理シフタ 3 の Verilog コード例

```
//
// XST does not infer a logical shifter for this example,
// as the value is not incremented by 1 for each consequent
// binary value of the selector.
//

module v_logical_shifters_3 (DI, SEL, SO);
    input [7:0] DI;
    input [1:0] SEL;
    output [7:0] SO;
    reg[7:0] SO;

    always @(DI or SEL)
    begin
        case (SEL)
            2'b00 : SO = DI;
            2'b01 : SO = DI << 1;
            2'b10 : SO = DI << 3;
            default : SO = DI << 2;
        endcase
    end
endmodule
```

四則演算ブロックの HDL コーディング手法

XST では、次の四則演算ブロックがサポートされています。

- ・ 次を含む加算器
 - キャリー イン
 - キャリー アウト
 - キャリー イン/アウト
- ・ 減算器
- ・ 加減算器
- ・ コンパレータ
 - =
 - /=
 - <
 - <=
 - >
 - >=
- ・ 乗算器
- ・ 除算器

XST では、次の符号付きおよび符号なしの四則演算ブロック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

VHDL でサポートされている符号付き/符号なし演算の詳細については、「[レジスタの HDL コーディング手法](#)」を参照してください。

XST では、次のブロックでリソースが共有されます。

- ・ 加算器
- ・ 減算器
- ・ 加算器/減算器
- ・ 乗算器

四則演算ブロックのログ ファイル

XST ログ ファイルには、認識された加算器、減算器、および加減算器のタイプおよびビット幅が示されます。

```
...
Synthesizing Unit <adder>.
  Related source file is arithmetic_operations_1.vhd.
  Found 8-bit adder for signal <sum>.
  Summary:
    inferred    1 Adder/Subtractor(s).
Unit <adder> synthesized.

=====
HDL Synthesis Report

Macro Statistics
# Adders/Subtractors      : 1
  8-bit adder              : 1
=====
```

四則演算ブロック関連の制約

- ・ [DSP48 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 HDL コーディング手法

次のデバイス ファミリでは、加算器と減算器を DSP48 リソースにインプリメントできます。

- ・ Virtex®-4
- ・ Virtex-5
- ・ Spartan®-3A DSP

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さらに複雑なマクロの 1 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、デフォルトで最大限のマクロ 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、加減算器のログ ファイル

```
Synthesizing Unit <v_adders_4>.
  Related source file is "v_adders_4.v".
  Found 8-bit adder carry in/out for signal <$addsub0000>.
  Summary:
    inferred 1 Adder/Subtractor(s).
Unit <v_adders_4> synthesized.
```

=====

HDL Synthesis Report

```
Macro Statistics
# Adders/Subtractors                : 1
8-bit adder carry in/out            : 1
=====
```

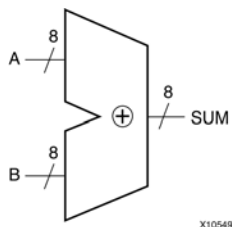
加算器、減算器、加減算器関連の制約

- ・ DSP48 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。アップデートは、
ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

符号なし 8 ビット加算器の図



符号なし 8 ビット加算器の IO ピンの説明

I/O ピン	説明
A、B	加算オペランド
SUM	加算結果

符号なし 8 ビット加算器の VHDL コード例

```
--
-- Unsigned 8-bit Adder
--

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;
```

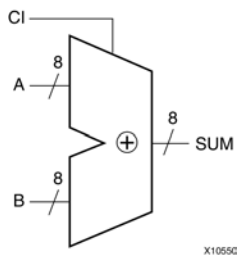
符号なし 8 ビット加算器の Verilog コード例

```
//
// Unsigned 8-bit Adder
//

module v_adders_1(A, B, SUM);
  input [7:0] A;
  input [7:0] B;
  output [7:0] SUM;

  assign SUM = A + B;
endmodule
```

キャリー イン付き符号なし 8 ビット加算器の図



キャリー イン付き符号なし 8 ビット加算器の IO ピンの説明

I/O ピン	説明
A、B	加算オペランド
CI	キャリー イン
SUM	加算結果

キャリー イン付き符号なし 8 ビット加算器の VHDL コード例

```
--  
-- Unsigned 8-bit Adder with Carry In  
--  
  
library ieee;  
use ieee.std_logic_1164.all;  
use ieee.std_logic_unsigned.all;  
  
entity adders_2 is  
  port(A,B : in std_logic_vector(7 downto 0);  
       CI : in std_logic;  
       SUM : out std_logic_vector(7 downto 0));  
end adders_2;  
  
architecture archi of adders_2 is  
begin  
  
  SUM <= A + B + CI;  
  
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  
//  
  
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
```

キャリー アウト付き符号なし 8 ビット加算器

VHDL の場合は、キャリー アウトのある加算演算子 (+) を記述する前に、使用する演算パッケージを確認してください。たとえば、std_logic_unsigned では、次の形式で「+」を使用してキャリー アウトを得ることはできません。

```
Res(9-bit) = A(8-bit) + B(8-bit)
```

これは、このパッケージで「+」を使用すると、加算結果のビット数が最も大きい引数（この場合は 8 ビット）と等しくなってしまうからです。

これを解決するには、連結を使用してオペランド A および B を 9 ビットに調整する方法があります。

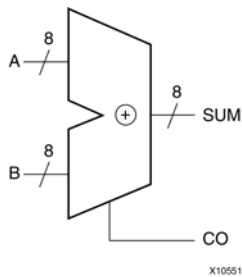
```
Res <= ("0" & A) + ("0" & B);
```

この場合、8 ビット加算器とキャリー アウトで 9 ビット加算器がインプリメントできると認識されます。

次は、別のソリューションです。

- ・ A と B を整数に変換します。
- ・ その結果を std_logic ベクタに変換し戻します。
- ・ ベクタのサイズを 9 に指定します。

キャリー アウト付き符号なし 8 ビット加算器の図



キャリー アウト付き符号なし 8 ビット加算器の IO ピンの説明

I/O ピン	説明
A, B	加算オペランド
SUM	加算結果
CO	キャリー アウト

キャリー アウト付き符号なし 8 ビット加算器の VHDL コード例

```
--
-- Unsigned 8-bit Adder with Carry Out
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity adders_3 is
  port(A,B : in std_logic_vector(7 downto 0);
        SUM : out std_logic_vector(7 downto 0);
        CO : out std_logic);
end adders_3;

architecture archi of adders_3 is
  signal tmp: std_logic_vector(8 downto 0);
begin

  tmp <= conv_std_logic_vector((conv_integer(A) + conv_integer(B)),9);
  SUM <= tmp(7 downto 0);
  CO <= tmp(8);

end archi;
```

上記の例では、次の 2 つの演算パッケージが使用されています。

- ・ **std_logic_arith**
このパッケージには、整数から std_logic への変換関数 (conv_std_logic_vector) が含まれています。
- ・ **std_logic_unsigned**
このパッケージには、符号なしの + (プラス) 演算が含まれています。

キャリー アウト付き符号なし 8 ビット加算器の Verilog コード例

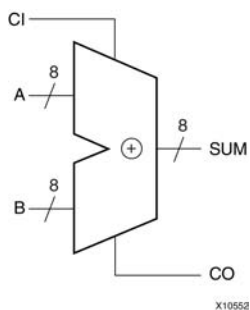
```
//
// Unsigned 8-bit Adder with Carry Out
//

module v_adders_3(A, B, SUM, CO);
  input [7:0] A;
  input [7:0] B;
  output [7:0] SUM;
  output CO;
  wire [8:0] tmp;

  assign tmp = A + B;
  assign SUM = tmp [7:0];
  assign CO = tmp [8];

endmodule
```

キャリー インおよびキャリー アウト付き符号なし 8 ビット加算器の図



キャリー イン およびキャリー アウト付き符号なし 8 ビット加算器の IO ピンの説明

I/O ピン	説明
A、B	加算オペランド
CI	キャリー イン
SUM	加算結果
CO	キャリー アウト

キャリー インおよびキャリー アウト付き符号なし 8 ビット加算器の VHDL コード例

```
--
-- Unsigned 8-bit Adder with Carry In and Carry Out
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity adders_4 is
  port(A,B : in std_logic_vector(7 downto 0);
       CI : in std_logic;
       SUM : out std_logic_vector(7 downto 0);
       CO : out std_logic);
end adders_4;

architecture archi of adders_4 is
  signal tmp: std_logic_vector(8 downto 0);
begin

  tmp <= conv_std_logic_vector((conv_integer(A) + conv_integer(B) + conv_integer(CI)),9);
  SUM <= tmp(7 downto 0);
  CO <= tmp(8);

end archi;
```

キャリー インおよびキャリー アウト付き符号なし 8 ビット加算器の Verilog コード例

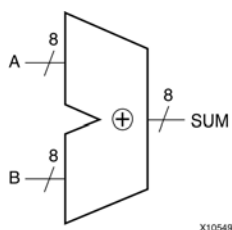
```
//
// Unsigned 8-bit Adder with Carry In and Carry Out
//

module v_adders_4(A, B, CI, SUM, CO);
  input CI;
  input [7:0] A;
  input [7:0] B;
  output [7:0] SUM;
  output CO;
  wire [8:0] tmp;

  assign tmp = A + B + CI;
  assign SUM = tmp [7:0];
  assign CO = tmp [8];

endmodule
```

符号付き 8 ビット加算器の図



符号付き 8 ビット加算器の IO ピンの説明

I/O ピン	説明
A、B	加算オペランド
SUM	加算結果

符号付き 8 ビット加算器の VHDL コード例

```
--
-- Signed 8-bit Adder
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity adders_5 is
  port(A,B : in std_logic_vector(7 downto 0);
        SUM : out std_logic_vector(7 downto 0));
end adders_5;

architecture archi of adders_5 is
begin

  SUM <= A + B;

end archi;
```

符号付き 8 ビット加算器の Verilog コード例

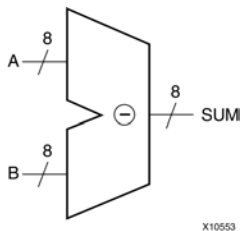
```
//
// Signed 8-bit Adder
//

module v_adders_5 (A,B,SUM);
  input signed [7:0] A;
  input signed [7:0] B;
  output signed [7:0] SUM;
  wire signed [7:0] SUM;

  assign SUM = A + B;

endmodule
```

符号なし 8 ビット減算器の図



符号なし 8 ビット減算器の IO ピンの説明

I/O ピン	説明
A、B	減算オペランド
RES	減算結果

符号なし 8 ビット減算器の VHDL コード例

```
--
-- Unsigned 8-bit Subtractor
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity adders_6 is
  port(A,B : in std_logic_vector(7 downto 0));
  RES : out std_logic_vector(7 downto 0));
end adders_6;

architecture archi of adders_6 is
begin

  RES <= A - B;

end archi;
```

符号なし 8 ビット減算器の Verilog コード例

```
//
// Unsigned 8-bit Subtractor
//

module v_adders_6(A, B, RES);
  input [7:0] A;
  input [7:0] B;
  output [7:0] RES;

  assign RES = A - B;

endmodule
```

ボロー イン付き符号なし 8 ビット減算器の IO ピンの説明

I/O ピン	説明
A, B	減算オペランド
BI	ボロー イン
RES	減算結果

ボロー イン付き符号なし 8 ビット減算器の VHDL コード例

```
--
-- Unsigned 8-bit Subtractor with Borrow In
--

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity adders_8 is
  port(A,B : in std_logic_vector(7 downto 0);
  BI : in std_logic;
  RES : out std_logic_vector(7 downto 0));
end adders_8;

architecture archi of adders_8 is
begin

  RES <= A - B - BI;

end archi;
```

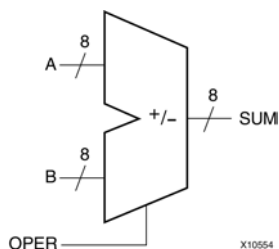
ボロー イン付き符号なし 8 ビット減算器の Verilog コード例

```
//
// Unsigned 8-bit Subtractor with Borrow In
//

module v_adders_8(A, B, BI, RES);
  input [7:0] A;
  input [7:0] B;
  input BI;
  output [7:0] RES;

  assign RES = A - B - BI;
endmodule
```

符号なし 8 ビット加減算器の図



符号なし 8 ビット加減算器の IO ピンの説明

I/O ピン	説明
A、B	比較オペランド
OPER	加算/減算選択入力
SUM	加算/減算結果

符号なし 8 ビット加減算器の VHDL コード例

```
--
-- Unsigned 8-bit Adder/Subtractor
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity adders_7 is
  port(A,B : in std_logic_vector(7 downto 0);
        OPER: in std_logic;
        RES : out std_logic_vector(7 downto 0));
end adders_7;

architecture archi of adders_7 is
begin

  RES <= A + B when OPER='0'
        else A - B;

end archi;
```

符号なし 8 ビット加減算器の Verilog コード例

```
//
// Unsigned 8-bit Adder/Subtractor
//

module v_adders_7(A, B, OPER, RES);
    input OPER;
    input [7:0] A;
    input [7:0] B;
    output [7:0] RES;
    reg [7:0] RES;

    always @(A or B or OPER)
    begin
        if (OPER==1'b0) RES = A + B;
        else RES = A - B;
    end
endmodule
```

コンパレータの HDL コーディング手法

このセクションには、次の内容が含まれます。

- ・ コンパレータのログ ファイル
- ・ コンパレータ関連の制約
- ・ コンパレータのコード例

コンパレータのログ ファイル

XST ログ ファイルには、認識されたコンパレータのタイプおよびビット幅が示されます。

```
...
Synthesizing Unit <compar>.
  Related source file is comparators_1.vhd.
  Found 8-bit comparator greatequal for signal <$n0000> created at line 10.
  Summary:
    inferred      1 Comparator(s).
Unit <compar> synthesized.

=====
HDL Synthesis Report

Macro Statistics
# Comparators           : 1
  8-bit comparator greatequal : 1
=====
...
```

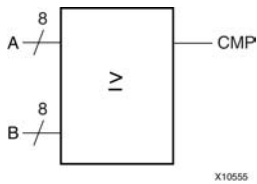
コンパレータ関連の制約

なし

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。アップデートは、
ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

符号なし 8 ビット コンパレータの図



符号なし 8 ビット コンパレータのピンの説明

I/O ピン	説明
A, B	比較オペランド
CMP	比較結果

符号なし 8 ビット コンパレータの VHDL コード例

```
--
-- Unsigned 8-bit Greater or Equal Comparator
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity comparator_1 is
    port(A,B : in  std_logic_vector(7 downto 0);
          CMP : out std_logic);
end comparator_1;

architecture archi of comparator_1 is
begin

    CMP <= '1' when A >= B else '0';

end archi;
```

符号なし 8 ビット コンパレータの Verilog コード例

```
//
// Unsigned 8-bit Greater or Equal Comparator
//

module v_comparator_1 (A, B, CMP);
    input  [7:0] A;
    input  [7:0] B;
    output CMP;

    assign CMP = (A >= B) ? 1'b1 : 1'b0;
endmodule
```

乗算器の HDL コーディング手法

乗算器をインプリメントする場合、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、A (8 ビット信号) と B (4 ビット信号) を掛け合わせる場合、結果は 12 ビット信号として宣言する必要があります。

レジスタ付き乗算器

次のデバイスでは、乗算器にレジスタを介した出力が必要な場合、特殊なレジスタ付き乗算器が推論されます。

- ・ Virtex®-4 デバイス
- ・ Virtex-5 デバイス

このレジスタ付き乗算器は 18 X 18 ビットです。

次の場合はレジスタ付き乗算器は使用されず、乗算器とレジスタが使用されます。

- ・ 乗算器の出力がレジスタ以外のコンポーネントに接続されている。
- ・ [乗算器スタイル \(MULT_STYLE\)](#) 制約が **lut** に設定されている。
- ・ 乗算器が非同期である。
- ・ 乗算器に同期リセットまたはクロック イネーブル以外の制御信号がある。
- ・ 乗算器が 1 つの 18 X 18 ブロック乗算器に収まらない。

レジスタ付き乗算器では、オプションで次のピンを使用できます。

- ・ クロック イネーブル ポート
- ・ 同期/非同期リセット ポートまたはロード ポート

乗算器

メモ： このセクションは、Virtex®-4、Virtex-5、Spartan®-3A DSP デバイスにのみ適用されます。

Virtex-4、Virtex-5、および Spartan-3A DSP デバイスでは、乗算器を DSP48 リソースにインプリメントできます。XST では、レジスタ付きのこれらのマクロをサポートし、DSP48 ブロックに 最大 2 段の入力レジスタと 2 段の出力レジスタを挿入できます。

乗算器のインプリメンテーションで複数の DSP48 リソースが必要な場合は、XST で乗算器が自動的に分解されて複数の DSP ブロックに含められます。オペランドのサイズによっては、最良の結果が得られるように、乗算器のほとんどの部分が DSP48 ブロックを使用してインプリメントされ、残りがスライス ロジックを使用してインプリメントされる場合があります。たとえば、18 X18 符号なし乗算器を 1 つインプリメントするには、DSP48 1 つでは不十分です。この場合は、このロジックのほとんどが DSP48 ブロックにインプリメントされ、残りが LUT にインプリメントされます。

Virtex-4、Virtex-5、Spartan-3A DSP デバイスでは、LUT を使用したインプリメンテーションに加え、DSP48 を使用したインプリメンテーションでもパイプライン乗算器を推論できます。詳細は、XST の制限についての記述を参照してください。

DSP48 ブロックへのマクロ インプリメンテーションは、[DSP48 の使用 \(USE_DSP48\)](#) 制約またはコマンドライン オプションでデフォルト auto に設定すると制御されます。このモードでは、アキュムレータがインプリメントされる際に、デバイス上で DSP48 リソースが考慮されます。

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 DSP48 リソースが制御されます。XST では、デフォルトで DSP48 リソースをすべて使用しようとします。詳細は、「[DSP48 ブロック リソース](#)」を参照してください。

XST では、値 lut および block が設定されている [乗算器スタイル \(MULT_STYLE\)](#) 制約が自動的に認識されて、これらが内部で [DSP48 の使用 \(USE_DSP48\)](#) に変換されます。Virtex-4 および Virtex-5 デザインで乗算器のインプリメンテーションに使用する FPGA リソースを定義するときは、[DSP48 の使用 \(USE_DSP48\)](#) 制約を使用してください。[乗算器スタイル \(MULT_STYLE\)](#) 制約は、選択した FPGA リソースで乗算器のインプリメンテーション方法を定義するときに使用してください。たとえば、[DSP48 の使用 \(USE_DSP48\)](#) が auto または yes に設定されていて、複数の DSP48 ブロックが必要な場合、mult_style=pipe_block を使用すると DSP48 のインプリメンテーションをパイプライン化できます。[DSP48 \(USE_DSP48\) の使用](#)が no に設定されている場合、mult_style=pipe_lut|KCM|CSD を使用して、LUT に乗算器をインプリメンテーションする方法を定義できます。

XST では、DSP48 に最大数のレジスタを含めるなど、デフォルトで最大限のマクロ 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、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。

定数との乗算

乗算に使用される引数の 1 つが定数の場合は、次の 2 つの方法を使用し、定数を使用した乗算という効率的な専用インプリメンテーションが作成されます。

- ・ KCM (Constant Coefficient Multiplier)
- ・ CSD (Canonical Signed Digit)

専用インプリメンテーションが、定数を使用した乗算器で必ずしも最良の結果になるとは限りません。XST では、KCM が通常の乗算インプリメンテーションのいずれかが自動的に選択されます。CSD は自動的に選択されません。CSD を使用する場合は、[MUX スタイル \(MUX_STYLE\)](#) 制約で指定します。

符号付きの数値を使用する場合は、KCM または CSD インプリメンテーションはサポートされません。

引数が 32 ビットを超える場合、[乗算器スタイル \(MULT_STYLE\)](#) 制約で指定されていても、KCM または CSD インプリメンテーションは使用されません。

乗算器のログ ファイル

XST ログ ファイルには、認識された乗算器のタイプおよびビット幅が示されます。

```
...
Synthesizing Unit <mult>.
  Related source file is multipliers_1.vhd.
  Found 8x4-bit multiplier for signal <res>.
  Summary:
    inferred    1 Multiplier(s).
  Unit <mult> synthesized.
```

```
=====
HDL Synthesis Report

Macro Statistics
# Multipliers                : 1
  8x4-bit multiplier          : 1
=====
...
```

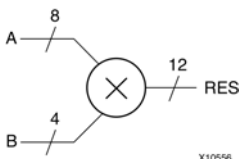
乗算器関連の制約

- ・ [乗算器スタイル \(MULT_STYLE\)](#)
- ・ [DSP48 の使用 \(USE_DSP48\)](#)
- ・ [DSP 使用率 \(DSP_UTILIZATION_RATIO\)](#)
- ・ [キープ \(KEEP\)](#)

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。アップデートは、http://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

符号なし 8 X 4 ビット乗算器の図



符号なし 8X4 ビット乗算器のピンの説明

I/O ピン	説明
A、B	乗算オペランド
RES	乗算結果

符号なし 8X4 ビット乗算器の VHDL コード例

```
--
-- Unsigned 8x4-bit Multiplier
--

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 8X4 ビット乗算器の Verilog コード例

```
//
// Unsigned 8x4-bit Multiplier
//

module v_multipliers_1(A, B, RES);
    input [7:0] A;
    input [3:0] B;
    output [11:0] RES;

    assign RES = A * B;
endmodule
```

逐次型複素乗算器の HDL コーディング手法

逐次型複素乗算器は、中間結果を累積して完全な乗算を行うのに 4 サイクル必要な複素乗算器です。インプリメンテーションには、DSP ブロックが 1 つ必要です。

2 つの複素数、A と B を乗算するには、次の 4 サイクルが必要です。

最初の 2 つのサイクルでは、次が計算されます。

$$\text{Res_real} = A_real * B_real - A_imag * B_imag$$

次の 2 つのサイクルでは、次が計算されます。

$$\text{Res_imag} = A_real * B_imag + A_imag * B_real$$

上記を実行するテンプレートはありますが、XST では DSP モードの違いを記述したり、enum 値を格納するのに enum 型または integer 型が使用できません。このため、基本的なテンプレートを使用して XST の推論を簡単にしてください。この一般的なアキュムレータのテンプレートを使用すると、XST では次を実行するために 1 つの DSP が推論されます。

- ・ **Load:** $P \leq \text{Value}$
- ・ **Load:** $P \leq -\text{Value}$
- ・ **Accumulate:** $P \leq P + \text{Value}$
- ・ **Accumulate:** $P \leq P - \text{Value}$

このテンプレートは、上記の 4 つの演算を実行する次の 2 つの制御信号で動作します。

- ・ **load**
- ・ **addsub**

逐次型複素乗算器のログ ファイル

なし

逐次型複素乗算器関連の制約

なし

逐次型複素乗算器のコード例

コード例は、本書が作成された時点のものです。アップデートは、
ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

符号付き 18X18 ビットの逐次型複素乗算器のピンの説明

I/O ピン	説明
CLK	クロック信号
Oper_Load、Oper_AddSub	load および addsub をコントロールする制御信号
A、B	乗算オペランド
RES	乗算結果

符号付き 18X18 ビットの逐次型複素乗算器の VHDL コード例

```
--
-- Sequential Complex Multiplier
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity multipliers_8 is
    generic(A_WIDTH:    positive:=18;
           B_WIDTH:    positive:=18;
           RES_WIDTH:   positive:=48);
    port(
        CLK:            in  std_logic;
        A:              in  signed(A_WIDTH-1 downto 0);
        B:              in  signed(B_WIDTH-1 downto 0);

        Oper_Load:      in  std_logic;
        Oper_AddSub:    in  std_logic;
        -- Oper_Load Oper_AddSub Operation
        -- 0          0          R= +A*B
    );
end entity;
```



```

-- 0      1      R= -A*B
-- 1      0      R=R+A*B
-- 1      1      R=R-A*B

    RES:          out signed(RES_WIDTH-1 downto 0)
);
end multipliers_8;

architecture beh of multipliers_8 is

    constant P_WIDTH: integer:=A_WIDTH+B_WIDTH;

    signal oper_load0: std_logic:='0';
    signal oper_addsub0: std_logic:='0';

    signal p1: signed(P_WIDTH-1 downto 0):=(others=>'0');
    signal oper_load1: std_logic:='0';
    signal oper_addsub1: std_logic:='0';

    signal res0: signed(RES_WIDTH-1 downto 0);
begin

    process (clk)
        variable acc: signed(RES_WIDTH-1 downto 0);
    begin
        if rising_edge(clk) then
            oper_load0  <= Oper_Load;
            oper_addsub0 <= Oper_AddSub;

            p1 <= A*B;
            oper_load1  <= oper_load0;
            oper_addsub1 <= oper_addsub0;

            if (oper_load1='1') then
                acc := res0;
            else
                acc := (others=>'0');
            end if;

            if (oper_addsub1='1') then
                res0 <= acc-p1;
            else
                res0 <= acc+p1;
            end if;

        end if;
    end process;

    RES <= res0;
end architecture;

```

符号付き 18X18 ビットの逐次型複素乗算器の Verilog コード例

```

module v_multipliers_8(CLK,A,B,Oper_Load,Oper_AddSub, RES);
    parameter A_WIDTH    = 18;
    parameter B_WIDTH    = 18;
    parameter RES_WIDTH  = 48;
    parameter P_WIDTH    = A_WIDTH+B_WIDTH;

    input  CLK;
    input  signed [A_WIDTH-1:0] A, B;

    input  Oper_Load, Oper_AddSub;
    // Oper_Load  Oper_AddSub  Operation
    //  0          0           R= +A*B
    //  0          1           R= -A*B
    //  1          0           R=R+A*B
    //  1          1           R=R-A*B

    output [RES_WIDTH-1:0] RES;

    reg oper_load0    = 0;
    reg oper_addsub0  = 0;

    reg signed [P_WIDTH-1:0] p1 = 0;
    reg oper_load1    = 0;
    reg oper_addsub1  = 0;

    reg signed [RES_WIDTH-1:0] res0 = 0;
    reg signed [RES_WIDTH-1:0] acc;

    always @(posedge CLK)
    begin
        oper_load0    <= Oper_Load;
        oper_addsub0  <= Oper_AddSub;

        p1 <= A*B;
        oper_load1    <= oper_load0;
        oper_addsub1  <= oper_addsub0;

        if (oper_load1==1'b1)
            acc = res0;
        else
            acc = 0;

        if (oper_addsub1==1'b1)
            res0 <= acc-p1;
        else
            res0 <= acc+p1;

    end

    assign RES = res0;
endmodule

```

パイプライン乗算器の HDL コーディング手法

XST では、大型乗算器を含むデザインでスピードを向上させるため、パイプライン乗算器を推論できます。大型乗算器の段の間にレジスタを挿入してパイプライン化することにより、デザイン全体の周波数を大幅に向上させることができます。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](#)」で説明されている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 コードで必要な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。

Virtex®-4 または Virtex-5 デバイスをターゲットにしている乗算器のインプリメンテーションに複数の DSP48 ブロックが必要な場合は、このインプリメンテーションもパイプライン化できます。このインスタンスの乗算器スタイル (MULT_STYLE) 制約を pipe_lut に設定します。

XST では最大の乗算器速度に到達させるため、次の両方の場合に使用可能なレジスタの最大数を使用します。

- ・ XST でパイプラインに有効なレジスタが検出される場合
- ・ 乗算器スタイル (MULT_STYLE) 制約が pipe_lut または pipe_block に設定される場合

XST では、各乗算器で周波数を最大にするために使用するレジスタの最大数が自動的に計算されます。

アドバンス HDL 合成段階中、XST HDL Advisor からは次の場合に最適なレジスタ ステージ数を指定するようにメッセージが表示されます。

- ・ まだ十分な数のレジスタ ステージを指定していない場合
- ・ 乗算器スタイル (MULT_STYLE) が信号に直接コード記述されている場合

XST では、次の場合に未使用のステージがシフトレジスタとしてインプリメントされます。

- ・ 乗算器の後に配置されたレジスタの数が必要な最大数を超える場合
- ・ シフトレジスタの抽出がオンになっている場合

XST には、次の制限があります。

- ・ XST では、ハードウェア乗算器 (MULT18X18S リソースを使用したインプリメンテーション) はパイプライン化できません。
- ・ レジスタに非同期セット/リセットまたは同期リセット信号が含まれる場合は、乗算器はパイプライン化できません。同期リセット信号が含まれる場合は、パイプライン化できます。

パイプライン乗算器のログ ファイル

次に、出力されるログ ファイルを示します。

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

Synthesizing Unit <multipliers_2>.
  Related source file is "multipliers_2.vhd".
  Found 36-bit register for signal <MULT>.
  Found 18-bit register for signal <a_in>.
  Found 18-bit register for signal <b_in>.
  Found 18x18-bit multiplier for signal <mult_res>.
  Found 36-bit register for signal <pipe_1>.
  Found 36-bit register for signal <pipe_2>.
  Found 36-bit register for signal <pipe_3>.
  Summary:
    inferred 180 D-type flip-flop(s).
    inferred 1 Multiplier(s).
Unit <multipliers_2> synthesized.
...
=====
*                               Advanced HDL Synthesis                               *
=====

Synthesizing (advanced) Unit <multipliers_2>.
Found pipelined multiplier on signal <mult_res>:
- 4 pipeline level(s) found in a register connected to the
multiplier macro output.
Pushing register(s) into the multiplier macro.
INFO:Xst - HDL ADVISOR - You can improve the performance of the
multiplier Mmult_mult_res by adding 1 register level(s).
Unit <multipliers_2> synthesized (advanced).

=====
HDL Synthesis Report

Macro Statistics
# Multipliers                : 1
 18x18-bit registered multiplier : 1
=====
```

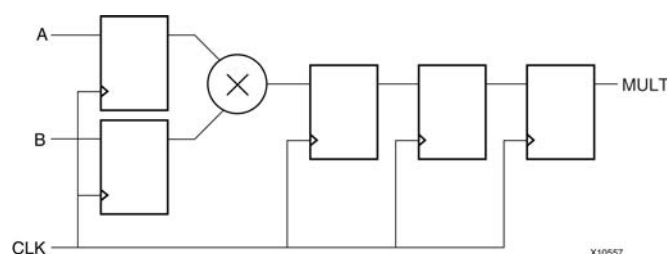
パイプライン乗算器関連の制約

- ・ [DSP48 の使用 \(USE_DSP48\)](#)
- ・ [DSP 使用率 \(DSP_UTILIZATION_RATIO\)](#)
- ・ [キープ \(KEEP\)](#)
- ・ [乗算器スタイル \(MULT_STYLE\)](#)

パイプライン乗算器のコード例

コード例は、本書が作成された時点のものです。アップデートは、
ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

パイプライン乗算器 (外部、単一レジスタ) の図



パイプライン乗算器 (外部、単一レジスタ) のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
A、B	乗算オペランド
MULT	乗算結果

パイプライン乗算器 (外部、単一レジスタ) の VHDL コード例

```
--
-- Pipelined multiplier
-- The multiplication operation placed outside the
-- process block and the pipeline stages represented
-- as single registers.
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity multipliers_2 is
    generic(A_port_size : integer := 18;
           B_port_size : integer := 18);
    port(clk : in std_logic;
         A : in unsigned (A_port_size-1 downto 0);
         B : in unsigned (B_port_size-1 downto 0);
         MULT : out unsigned ( (A_port_size+B_port_size-1) downto 0));

    attribute mult_style: string;
    attribute mult_style of multipliers_2: entity is "pipe_lut";
end multipliers_2;

architecture beh of multipliers_2 is
    signal a_in, b_in : unsigned (A_port_size-1 downto 0);
    signal mult_res : unsigned ( (A_port_size+B_port_size-1) downto 0);
    signal pipe_1,
           pipe_2,
           pipe_3 : unsigned ((A_port_size+B_port_size-1) downto 0);
begin

    mult_res <= a_in * b_in;

    process (clk)
    begin
        if (clk'event and clk='1') then
            a_in <= A; b_in <= B;
            pipe_1 <= mult_res;
            pipe_2 <= pipe_1;
            pipe_3 <= pipe_2;
            MULT <= pipe_3;
        end if;
    end process;
end beh;
```

パイプライン乗算器 (外部、単一レジスタ) の Verilog コード例

```
//
// Pipelined multiplier
// The multiplication operation placed outside the
// always block and the pipeline stages represented
// as single registers.
//

(*mult_style="pipe_lut"*)
module v_multipliers_2(clk, A, B, MULT);

    input clk;
    input [17:0] A;
    input [17:0] B;
    output [35:0] MULT;
    reg [35:0] MULT;
    reg [17:0] a_in, b_in;
    wire [35:0] mult_res;
    reg [35:0] pipe_1, pipe_2, pipe_3;

    assign mult_res = a_in * b_in;

    always @(posedge clk)
    begin
        a_in <= A; b_in <= B;
        pipe_1 <= mult_res;
        pipe_2 <= pipe_1;
        pipe_3 <= pipe_2;
        MULT <= pipe_3;
    end
endmodule
```

パイプライン乗算器 (内部、単一レジスタ) のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
A, B	乗算オペランド
MULT	乗算結果

パイプライン乗算器 (内部、単一レジスタ) の VHDL コード例

```
--
-- Pipelined multiplier
-- The multiplication operation placed inside the
-- process block and the pipeline stages represented
-- as single registers.
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity multipliers_3 is
    generic(A_port_size: integer := 18;
           B_port_size: integer := 18);
    port(clk : in std_logic;
         A : in unsigned (A_port_size-1 downto 0);
         B : in unsigned (B_port_size-1 downto 0);
         MULT : out unsigned ((A_port_size+B_port_size-1) downto 0));

    attribute mult_style: string;
    attribute mult_style of multipliers_3: entity is "pipe_lut";

end multipliers_3;

architecture beh of multipliers_3 is
    signal a_in, b_in : unsigned (A_port_size-1 downto 0);
    signal mult_res : unsigned ((A_port_size+B_port_size-1) downto 0);
    signal pipe_2,
           pipe_3 : unsigned ((A_port_size+B_port_size-1) downto 0);

begin
    process (clk)
    begin
        if (clk'event and clk='1') then
            a_in <= A; b_in <= B;
            mult_res <= a_in * b_in;
            pipe_2 <= mult_res;
            pipe_3 <= pipe_2;
            MULT <= pipe_3;
        end if;
    end process;
end beh;
```

パイプライン乗算器 (内部、単一レジスタ) の Verilog コード例

```
//  
// Pipelined multiplier  
// The multiplication operation placed inside the  
// process block and the pipeline stages are represented  
// as single registers.  
//  
  
(*mult_style="pipe_lut"*)  
module v_multipliers_3(clk, A, B, MULT);  
  
    input clk;  
    input [17:0] A;  
    input [17:0] B;  
    output [35:0] MULT;  
    reg [35:0] MULT;  
    reg [17:0] a_in, b_in;  
    reg [35:0] mult_res;  
    reg [35:0] pipe_2, pipe_3;  
  
    always @(posedge clk)  
    begin  
        a_in <= A; b_in <= B;  
        mult_res <= a_in * b_in;  
        pipe_2 <= mult_res;  
        pipe_3 <= pipe_2;  
        MULT <= pipe_3;  
    end  
endmodule
```

パイプライン乗算器 (外部、シフトレジスタ) のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
A, B	乗算オペランド
MULT	乗算結果

パイプライン乗算器 (外部、シフトレジスタ) の VHDL コード例

```
--
-- Pipelined multiplier
-- The multiplication operation placed outside the
-- process block and the pipeline stages represented
-- as shift registers.
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity multipliers_4 is
    generic(A_port_size: integer := 18;
           B_port_size: integer := 18);
    port(clk : in std_logic;
         A : in unsigned (A_port_size-1 downto 0);
         B : in unsigned (B_port_size-1 downto 0);
         MULT : out unsigned ( (A_port_size+B_port_size-1) downto 0));

    attribute mult_style: string;
    attribute mult_style of multipliers_4: entity is "pipe_lut";

end multipliers_4;

architecture beh of multipliers_4 is
    signal a_in, b_in : unsigned (A_port_size-1 downto 0);
    signal mult_res : unsigned ((A_port_size+B_port_size-1) downto 0);

    type pipe_reg_type is array (2 downto 0) of unsigned ((A_port_size+B_port_size-1) downto 0);
    signal pipe_regs : pipe_reg_type;

begin

    mult_res <= a_in * b_in;

    process (clk)
    begin
        if (clk'event and clk='1') then
            a_in <= A; b_in <= B;
            pipe_regs <= mult_res & pipe_regs(2 downto 1);
            MULT <= pipe_regs(0);
        end if;
    end process;
end beh;
```

パイプライン乗算器 (外部、シフトレジスタ) の Verilog コード例

```
//  
// Pipelined multiplier  
// The multiplication operation placed outside the  
// always block and the pipeline stages represented  
// as shift registers.  
//  
  
(*mult_style="pipe_lut"*)  
module v_multipliers_4(clk, A, B, MULT);  
  
    input clk;  
    input [17:0] A;  
    input [17:0] B;  
    output [35:0] MULT;  
    reg [35:0] MULT;  
    reg [17:0] a_in, b_in;  
    wire [35:0] mult_res;  
    reg [35:0] pipe_regs [2:0];  
    integer i;  
  
    assign mult_res = a_in * b_in;  
  
    always @(posedge clk)  
    begin  
        a_in <= A; b_in <= B;  
  
        pipe_regs[2] <= mult_res;  
        for (i=0; i<=1; i=i+1) pipe_regs[i] <= pipe_regs[i+1];  
  
        MULT <= pipe_regs[0];  
    end  
endmodule
```

乗算/加減算器の HDL コーディング手法

乗算/加減算器は、次のような複数の基本マクロから構成されている複雑なマクロです。

- ・ 乗算器
- ・ 加算器/減算器
- ・ レジスタ

このマクロは、次のデバイスの DSP48 リソースにインプリメントできます。

- ・ Virtex®-4
- ・ Virtex-5

Virtex®-4 および Virtex-5 デバイスの乗算/加減算器のコーディング手法

XST では、レジスタ付きのこのマクロがサポートされており、乗算器の入力に付いている最大 2 段の入力レジスタと加減算器の入力に付いている 1 段の出力レジスタを DSP48 ブロックに挿入できます。キャリーインまたは加減算のセレクタ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。また、乗算処理にもレジスタを付けることができます。

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 つのみの場合に乗算/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、乗算器と加減算器が別々のマクロとして処理されます。詳細は、「乗算器の HDL コーディング手法」および「加算器、減算器、加減算器の HDL コーディング手法」を参照してください。

DSP48 ブロックへのマクロ インプリメンテーションは、[DSP48 の使用 \(USE_DSP48\)](#) 制約またはコマンドライン オプションでデフォルト auto に設定すると制御されます。このモードでは、乗算/加減算器がインプリメントされる際に、デバイス上で DSP48 リソースが考慮されます。

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 DSP48 リソースが制御されます。XST では、デフォルトで DSP48 リソースをすべて使用しようとします。詳細は、「[DSP48 ブロック リソース](#)」を参照してください。

XST では、DSP48 に最大数のレジスタを含めるなど、デフォルトで最大限のマクロ 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、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。

ログ ファイルには、HDL 合成段階で推論された乗算器、加算器、減算器、およびレジスタの詳細がレポートされています。推論された MAC の情報は、MAC のインプリメンテーションの起こるアドバンス HDL 合成中にレポートされます。

乗算/加減算器のログ ファイル

ログ ファイルには、HDL 合成段階で推論された乗算器、加減算器、およびレジスタの詳細がレポートされています。乗算/加減算器は、アドバンス HDL 合成段階で作成されます。これらは MAC のインプリメンテーションに含まれるので、ログ ファイルには推論された MAC についての情報が表示されます。

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

Synthesizing Unit <multipliers_6>.
  Related source file is "multipliers_6.vhd".
  Found 8-bit register for signal <A_reg1>.
  Found 8-bit register for signal <A_reg2>.
  Found 8-bit register for signal <B_reg1>.
  Found 8-bit register for signal <B_reg2>.
  Found 8x8-bit multiplier for signal <mult>.
  Found 16-bit addsub for signal <multaddsub>.
  Summary:
    inferred 32 D-type flip-flop(s).
    inferred 1 Adder/Subtractor(s).
    inferred 1 Multiplier(s).
Unit <multipliers_6> synthesized.
...
=====
*                               Advanced HDL Synthesis                               *
=====

...
Synthesizing (advanced) Unit <Mmult_mult>.
  Multiplier <Mmult_mult> in block <multipliers_6> and adder/subtractor
  <Maddsub_multaddsub> in block <multipliers_6> are combined into a
  MAC<Mmac_Maddsub_multaddsub>.
  The following registers are also absorbed by the MAC: <A_reg2> in block
  <multipliers_6>, <A_reg1> in block <multipliers_6>, <B_reg2> in
  block <multipliers_6>, <B_reg1> in block <multipliers_6>.
Unit <Mmult_mult> synthesized (advanced).

=====
HDL Synthesis Report

=====

Macro Statistics
# MACs                               : 1
8x8-to-16-bit MAC                     : 1
=====
```

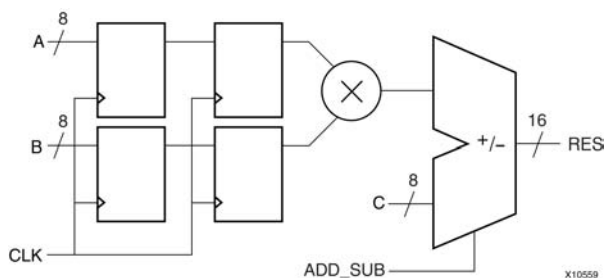
乗算/加減算器関連の制約

- ・ DSP48 の使用 (USE_DSP48)
- ・ DSP 使用率 (DSP_UTILIZATION_RATIO)
- ・ キープ (KEEP)

乗算/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。アップデートは、
ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

乗算器の入力に 2 段のレジスタが付いた乗算/加算器の図



乗算器の入力に 2 段のレジスタが付いた乗算/加算器のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
A、B、C	乗算/加算オペランド
RES	乗算/加算結果

乗算器の入力に 2 段のレジスタが付いた乗算/加算器の VHDL コード例

```
--
-- Multiplier Adder with 2 Register Levels on Multiplier Inputs
--

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity multipliers_5 is
    generic (p_width: integer:=8);
    port (clk : in std_logic;
          A, B, C : in std_logic_vector(p_width-1 downto 0);
          RES : out std_logic_vector(p_width*2-1 downto 0));
end multipliers_5;

architecture beh of multipliers_5 is
    signal A_reg1, A_reg2,
           B_reg1, B_reg2 : std_logic_vector(p_width-1 downto 0);
    signal multaddsub : std_logic_vector(p_width*2-1 downto 0);
begin

    multaddsub <= A_reg2 * B_reg2 + C;

    process (clk)
    begin
        if (clk'event and clk='1') then
            A_reg1 <= A; A_reg2 <= A_reg1;
            B_reg1 <= B; B_reg2 <= B_reg1;
        end if;
    end process;

    RES <= multaddsub;

end beh;
```

乗算器の入力に 2 段のレジスタが付いた乗算/加算器の Verilog コード例

```
//
// Multiplier Adder with 2 Register Levels on Multiplier Inputs
//

module v_multipliers_5 (clk, A, B, C, RES);

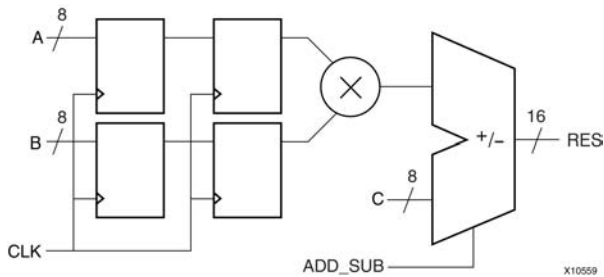
    input  clk;
    input  [7:0] A;
    input  [7:0] B;
    input  [7:0] C;
    output [15:0] RES;
    reg    [7:0] A_reg1, A_reg2, B_reg1, B_reg2;
    wire   [15:0] multaddsub;

    always @(posedge clk)
    begin
        A_reg1 <= A; A_reg2 <= A_reg1;
        B_reg1 <= B; B_reg2 <= B_reg1;
    end

    assign multaddsub = A_reg2 * B_reg2 + C;
    assign RES = multaddsub;

endmodule
```

乗算器の入力に 2 段のレジスタが付いた乗算/加減算器の図



乗算器の入力に 2 段のレジスタが付いた乗算/加減算器のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
ADD_SUB	加減算セクタ
A、B、C	乗算/加減算オペランド
RES	乗算/加減算結果

乗算器の入力に 2 段のレジスタが付いた乗算/加減算器の VHDL コード例

```
--
-- Multiplier Adder/Subtractor with
-- 2 Register Levels on Multiplier Inputs
--

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity multipliers_6 is
    generic (p_width: integer:=8);
    port (clk,add_sub: in std_logic;
          A, B, C: in std_logic_vector(p_width-1 downto 0);
          RES: out std_logic_vector(p_width*2-1 downto 0));
end multipliers_6;

architecture beh of multipliers_6 is
    signal A_reg1, A_reg2,
           B_reg1, B_reg2 : std_logic_vector(p_width-1 downto 0);
    signal mult, multaddsub : std_logic_vector(p_width*2-1 downto 0);
begin

    mult <= A_reg2 * B_reg2;
    multaddsub <= C + mult when add_sub = '1' else C - mult;

    process (clk)
    begin
        if (clk'event and clk='1') then
            A_reg1 <= A; A_reg2 <= A_reg1;
            B_reg1 <= B; B_reg2 <= B_reg1;
        end if;
    end process;

    RES <= multaddsub;

end beh;
```

乗算器の入力に 2 段のレジスタが付いた乗算/加減算器の Verilog コード例

```
//
// Multiplier Adder/Subtractor with
// 2 Register Levels on Multiplier Inputs
//

module v_multipliers_6 (clk, add_sub, A, B, C, RES);

    input  clk, add_sub;
    input  [7:0] A;
    input  [7:0] B;
    input  [7:0] C;
    output [15:0] RES;
    reg    [7:0] A_reg1, A_reg2, B_reg1, B_reg2;
    wire   [15:0] mult, multaddsub;

    always @(posedge clk)
    begin
        A_reg1 <= A; A_reg2 <= A_reg1;
        B_reg1 <= B; B_reg2 <= B_reg1;
    end

    assign mult = A_reg2 * B_reg2;
    assign multaddsub = add_sub ? C + mult : C - mult;
    assign RES = multaddsub;

endmodule
```

MAC の HDL コーディング手法

MAC (積和演算) は、次のような複数の基本マクロから構成されている複雑なマクロです。

- ・ 乗算器
- ・ アキュムレータ
- ・ レジスタ

このマクロは、次のデバイスの DSP48 リソースにインプリメントできます。

- ・ Virtex®-4
- ・ Virtex-5

Virtex-4 および Virtex-5 デバイスの MAC

MAC (積和演算) は、乗算器、アキュムレータ、およびレジスタなどの複数の基本マクロから構成されている複雑なマクロです。このマクロは、Virtex®-4 および Virtex-5 デバイスの DSP48 リソースにインプリメントできます。

XST では、レジスタ付きのこのマクロがサポートされており、最大 2 段の入力レジスタを DSP48 ブロックに挿入できます。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。また、乗算処理にもレジスタを付けることができます。

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 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。詳細は、「[乗算器の HDL コーディング手法](#)」および「[アキュムレータの HDL コーディング手法](#)」を参照してください。

DSP48 ブロックへのマクロ インプリメンテーションは、[DSP48 の使用 \(USE_DSP48\)](#) 制約またはコマンドライン オプションでデフォルト auto に設定すると制御されます。このモードでは、MAC がインプリメントされる際に、デバイス上で DSP48 リソースが考慮されます。

また、auto モードの場合は、[DSP 使用率 \(DSP_UTILIZATION_RATIO\)](#) 制約を使用して DSP48 リソースを制御します。XST は、できるだけ多くの DSP48 リソースを使用しようとします。詳細は、「[DSP48 ブロック リソース](#)」を参照してください。

XST では、DSP48 に最大数のレジスタを含めるなど、デフォルトで最大限のマクロ 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ています。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、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。

ログ ファイルには、HDL 合成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ています。MAC は、アドバンス HDL 合成段階で作成されます。

MAC のログ ファイル

ログ ファイルには、HDL 合成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ています。MAC は、アドバンス HDL 合成段階で作成されます。

```
=====
*                               HDL Synthesis                               *
=====
...
Synthesizing Unit <multipliers_7a>.
  Related source file is "multipliers_7a.vhd".
  Found 8x8-bit multiplier for signal <$n0002> created at line 28.
  Found 16-bit up accumulator for signal <accum>.
  Found 16-bit register for signal <mult>.
  Summary:
    inferred   1 Accumulator(s).
    inferred  16 D-type flip-flop(s).
    inferred   1 Multiplier(s).
Unit <multipliers_7a> synthesized....
=====
*                               Advanced HDL Synthesis                       *
=====
...
Synthesizing (advanced) Unit <Mmult__n0002>.
  Multiplier <Mmult__n0002> in block <multipliers_7a> and accumulator
  <accum> in block <multipliers_7a> are combined into a MAC<Mmac_accum>.
  The following registers are also absorbed by the MAC: <mult> in block
  <multipliers_7a>. Unit <Mmult__n0002> synthesized (advanced).

=====
HDL Synthesis Report

Macro Statistics
# MACs                               : 1
 8x8-to-16-bit MAC                   : 1

=====
```

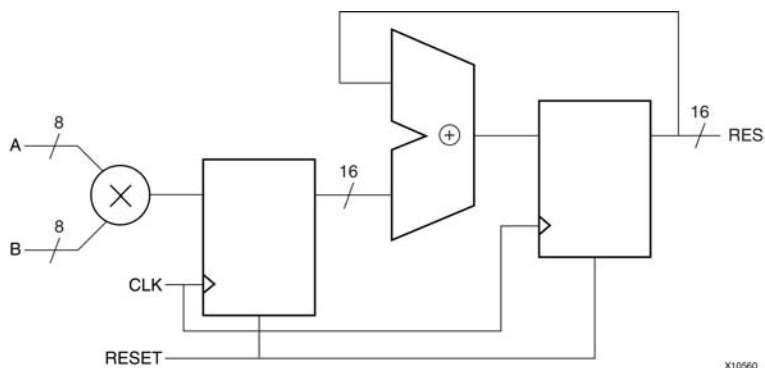
MAC 関連の制約

- ・ **DSP48 の使用 (USE_DSP48)**
- ・ **DSP 使用率 (DSP_UTILIZATION_RATIO)**
- ・ **キープ (KEEP)**

MAC 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。アップデートは、
ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

乗算アップ アキュムレータ (乗算後にレジスタ付き) の図



乗算アップ アキュムレータ (乗算後にレジスタ付き) のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
RESET	同期リセット
A、B	MAC オペランド
RES	MAC 結果

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
--
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;
```

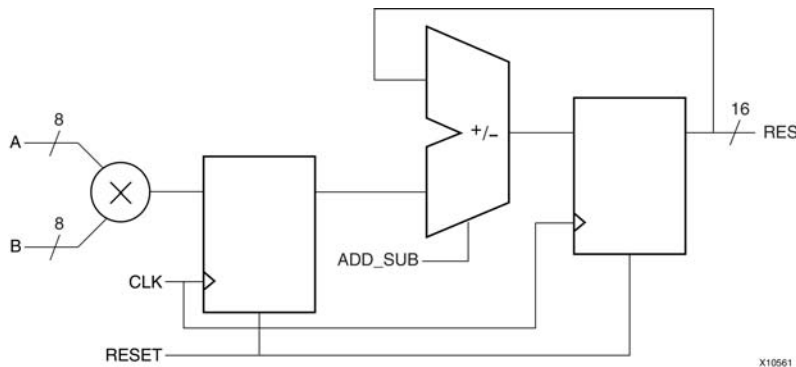
乗算アップ アキュムレータ (乗算後にレジスタ付き) の Verilog コード例

```
//
// Multiplier Up Accumulate with Register After Multiplication
//
module v_multipliers_7a (clk, reset, A, B, RES);
  input clk, reset;
  input [7:0] A;
  input [7:0] B;
  output [15:0] RES;
  reg [15:0] mult, accum;

  always @(posedge clk)
  begin
    if (reset)
      mult <= 16'b0000000000000000;
    else
      mult <= A * B;
    end

  always @(posedge clk)
  begin
    if (reset)
      accum <= 16'b0000000000000000;
    else
      accum <= accum + mult;
    end
    assign RES = accum;
  end
endmodule
```

乗算アップ/ダウン アキュムレータ (乗算後にレジスタ付き) の図



乗算アップ/ダウン アキュムレータ (乗算後にレジスタ付き) のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
RESET	同期リセット
ADD_SUB	加減算セレクト
A, B	MAC オペランド
RES	MAC 結果

乗算アップ/ダウン 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/Down Accumulate with Register After Multiplication.
--

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity multipliers_7b is
    generic (p_width: integer:=8);
    port (clk, reset, add_sub: 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_7b;

architecture beh of multipliers_7b 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

                if (add_sub = '1') then
                    accum <= accum + mult;
                else
                    accum <= accum - mult;
                end if;

                mult <= A * B;
            end if;
        end if;
    end process;

    RES <= accum;

end beh;
```

乗算アップ/ダウン アキュムレータ (乗算後にレジスタ付き) の Verilog コード例

```
//  
// Multiplier Up/Down Accumulate with Register After Multiplication.  
//  
module v_multipliers_7b (clk, reset, add_sub, A, B, RES);  
  
    input  clk, reset, add_sub;  
    input  [7:0] A;  
    input  [7:0] B;  
    output [15:0] RES;  
    reg    [15:0] mult, accum;  
  
    always @(posedge clk)  
    begin  
        if (reset)  
            mult <= 16'b0000000000000000;  
        else  
            mult <= A * B;  
        end  
  
    always @(posedge clk)  
    begin  
        if (reset)  
            accum <= 16'b0000000000000000;  
        else  
            if (add_sub)  
                accum <= accum + mult;  
            else  
                accum <= accum - mult;  
            end  
        end  
  
    assign RES = accum;  
  
endmodule
```

除算器の HDL コーディング手法

除算器は、序数が定数で 2 のべき乗の場合にのみサポートされます。この場合、演算子がシフタとしてインプリメントされます。それ以外の場合、XST でエラー メッセージが表示されます。

除算器のログ ファイル

除数が定数で 2 のべき乗の除算器は、マクロ認識の段階ではレポートされません。除算器が XST でサポートされる文字に対応していない場合は、次のエラー メッセージが表示されます。

```
...  
ERROR:Xst:719 - file1.vhd (Line 172).  
Operator is not supported yet : 'DIVIDE'  
...
```

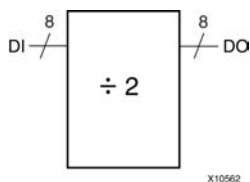
除算器関連の制約

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。アップデートは、
ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

定数 2 で割る除算器の図



定数 2 で割る除算器のピンの説明

I/O ピン	説明
DI	除算オペランド
DO	除算結果

定数 2 で割る除算器の VHDL コード例

```
--
-- Division By Constant 2
--

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 で割る除算器の Verilog コード例

```
//
// Division By Constant 2
//

module v_divider_1 (DI, DO);
    input  [7:0] DI;
    output [7:0] DO;

    assign DO = DI / 2;
endmodule
```

リソース共有の HDL コーディング手法

リソースの共有は、演算の数および合成されたデザインに含まれるロジックの数を最小限に抑えるために行われます。この最適化では、2 つの類似する演算リソースが同時に使用されない場合に、この 2 つを 1 つの演算子としてインプリメントします。XST では、リソースの共有と、必要に応じて、マルチプレクサの数を減らす処理が実行されます。

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
- ・ 加算器/減算器
- ・ 乗算器

最適化の目標がスピードである場合は、リソースの共有をオフにした方が良い結果が得られる場合があります。XST では、クロック周波数を向上するために、アドバンス HDL 合成段階でリソースの共有をオフにすることを推奨するメッセージが表示されます。

リソース共有のログ ファイル

XST ログ ファイルには、認識された数値演算ブロックおよび乗算器のタイプおよびビット幅が示されます。

```
...
Synthesizing Unit <addsub>.
  Related source file is resource_sharing_1.vhd.
  Found 8-bit addsub for signal <res>.
  Found 8 1-bit 2-to-1 multiplexers.
  Summary:
    inferred    1 Adder/Subtractor(s).
    inferred    8 Multiplexer(s).
  Unit <addsub> synthesized.

=====
HDL Synthesis Report

Macro Statistics
# Multiplexers                : 1
  2-to-1 multiplexer          : 1
# Adders/Subtractors          : 1
  8-bit addsub                : 1
=====
...
=====
*                               Advanced HDL Synthesis                               *
=====

INFO:Xst - 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.
...
```

リソース共有関連の制約

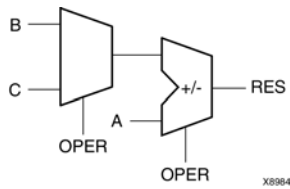
リソース共有 (RESOURCE_SHARING)

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。アップデートは、
http://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

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



リソース共有のピンの説明

I/O ピン	説明
A、B、C	オペランド
OPER	演算セレクト
RES	データ出力

リソース共有の VHDL コード例

```
--
-- Resource Sharing
--

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
//

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/ROM の HDL コーディング手法

HDL コードをアーキテクチャに依存しないようにするために、RAM プリミティブをインスタンス化しない場合は、XST の RAM の自動推論を使用してください。XST では、分散 RAM およびブロック RAM の両方を推論可能で、どちらの RAM でも次の機能がサポートされます。

- ・ 同期書き込み
- ・ 書き込みイネーブル
- ・ RAM イネーブル
- ・ 非同期または同期読み出し
- ・ データ出力ラッチのリセット
- ・ データ出力リセット
- ・ シングル ポート、デュアル ポート、または複数ポートリード
- ・ シングル ポート RAM およびデュアル ポートライト
- ・ パリティ ビット
- ・ バイト幅の書き込みイネーブル付きブロック RAM
- ・ シンプル デュアル ポート BRAM

負のアドレスが設定された RAM および ROM はサポートされません。

推論される RAM のタイプは、コードの記述方法によって異なります。

- ・ 非同期読み出しが含まれる RAM 記述では、分散 RAM マクロが推論されます。
- ・ 同期読み出しが含まれる RAM 記述では、ブロック RAM マクロが推論されます。ブロック RAM マクロが、実際には分散 RAM マクロを使用してインプリメントされる場合もあります。実際の RAM インプリメンテーションは、マクロ生成機能で決定されます。

テンプレートでブロック RAM と分散 RAM のどちらでもインプリメントできる場合は、ブロック RAM がインプリメントされます。[RAM スタイル \(RAM_STYLE\)](#) 制約を使用すると、RAM のインプリメンテーションを制御して必要な RAM タイプを選択できます。詳細は、「[デザイン制約](#)」を参照してください。

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
- ・ 各ポートに異なるワード数とビット幅の比率を使用
- ・ シンプル デュアル ポート分散 RAM
- ・ クワッド ポート分散 RAM

XST では、BRAM リソースに RAM が速度重視でインプリメントされるため、速度に関しては良い結果になるものの、エリア重視のインプリメンテーションよりも BRAM リソースが多く必要になってしまいます。XST では、エリア重視の BRAM 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、「[FPGA の最適化](#)」を参照してください。

XST では、次が実行できます。

- ・ Finite State Machine (FSM) コンポーネントをインプリメントします。詳細は、「[Finite State Machine \(FSM\) の HDL コーディング手法](#)」を参照してください。
- ・ 一般的なロジックをブロック RAM 上にマップします。詳細は、「[ブロック RAM へのロジックのマップ](#)」を参照してください。

XST では、ターゲット デバイスの BRAM リソースが自動的に制御されます。[BRAM 使用率 \(BRAM_UTILIZATION_RATIO\)](#) を使用すると、合成中に XST で処理される BRAM ブロック数を制限できます。

XST では、専用のリソースを使用して小型の RAM および ROM をインプリメントすることで、デザイン速度を改善します。RAM および ROM は、サイズが次の表の規則に従っている場合は小型のメモリと考えられます。

小型の RAM/ROM の条件

デバイス	サイズ (ビット) * 幅 (ビット)
Virtex®-4	<= 512
Virtex-5	<= 512

BRAM 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 \(RAM_STYLE\)](#) および [ROM スタイル \(ROM_STYLE\)](#) を使用してください。

XST では、次の計算式で、推論に使用可能な BRAM リソースの数が計算されます。

Total_Number_of_Available_BRAMs - Number_of_Reserved_BRAMs

この式では、それぞれ次を表しています。

Total_Number_of_Available_BRAMs

[BRAM 使用率 \(BRAM_UTILIZATION_RATIO\)](#) 制約で指定した BRAM 数になります。デフォルトは 100% です。

Number of Reserved_BRAMs は、次の合計です。

- ・ UNISIM ライブラリからの Hardware Description Language (HDL) コードに含まれるインスタンスエート済みの BRAM 数
- ・ [RAM スタイル \(RAM_STYLE\)](#) および [ROM スタイル \(ROM_STYLE\)](#) 制約により BRAM として強制的にインプリメントされた RAM の数
- ・ BRAM マッピング最適化 (BRAM_MAP) 制約を使用して生成される BRAM の数

XST では、使用可能な BRAM リソースがある箇所に BRAM を使用して推論された最大の RAM および ROM をインプリメントし、分散リソースに最小の RAM および ROM をインプリメントします。

Number_of_Reserved_BRAMs の数が使用可能なリソースを超えると、それらがブロック RAM としてインプリメントされ、推論された RAM はすべて分散メモリにインプリメントされます。

このプロセスが終了した直後、2 つの小型のシングルポート BRAM が 1 つの BRAM プリミティブに自動的にパックされます。これは、[自動 BRAM パッキング \(AUTO_BRAM_PACKING\)](#) 制約で制御されます。この制約は、デフォルトではオフになっています。

詳細は、「[BRAM 使用率 \(BRAM_UTILIZATION_RATIO\)](#)」および「[自動 BRAM パッキング \(AUTO_BRAM_PACKING\)](#)」を参照してください。

RAM/ROM のログ ファイル

XST ログ ファイルには、認識された RAM のタイプ、サイズ、I/O ポートの情報が示されます。RAM の認識は、次の 2 つの段階から構成されています。

- ・ XST では、HDL 合成段階中に HDL コードにあるメモリ構造が認識されます。
- ・ アドバンス合成段階では、特定のメモリのインプリメンテーション方法 (ブロック型と分散型のどちらのメモリ リソースを使用するか) が決定されます。

=====

* HDL Synthesis *

=====

```
Synthesizing Unit <rams_16>.
Related source file is "rams_16.vhd".
Found 64x16-bit dual-port RAM <Mram_RAM> for signal <RAM>.
Found 16-bit register for signal <doa>.
Found 16-bit register for signal <dob>.
Summary:
inferred 1 RAM(s).
inferred 32 D-type flip-flop(s).
Unit <rams_16> synthesized.
```

=====

HDL Synthesis Report

Macro Statistics

```
# RAMs : 1
  64x16-bit dual-port RAM : 1
# Registers: 2
  16-bit register : 2
```

=====

=====

* Advanced HDL Synthesis*

=====

```
Synthesizing (advanced) Unit <rams_16>.
INFO:Xst - The RAM <Mram_RAM> will be implemented as a BLOCK RAM, absorbing
the following register(s): <doa> <dob>
```

ram_type	Block		

Port A			
aspect ratio	64-word x 16-bit		
mode	write-first		
clkA	connected to signal <clkA>	rise	
enA	connected to signal <ena>	high	
weA	connected to internal <wea>	high	
addrA	connected to signal <addrA>		
diA	connected to internal <dia>		
doA	connected to signal <doa>		

optimization	speed		
=====			

ram_type	Block		

```

-----
| Port B                                     |
|   aspect ratio   | 64-word x 16-bit           |           |
|   mode           | write-first                |           |
|   clkB           | connected to signal <clkb>   | rise      |
|   enB            | connected to signal <enb>    | high      |
|   weB            | connected to internal <web>  | high      |
|   addrB          | connected to signal <addrb>  |           |
|   diB            | connected to internal <dib>  |           |
|   doB            | connected to signal <dob>    |           |
|-----|-----|
| optimization     | speed                       |           |
|-----|-----|
=====
Unit <rams_16> synthesized (advanced).

=====
Advanced HDL Synthesis Report

Macro Statistics
# RAMs : 1
  64x16-bit dual-port block RAM : 1

=====

```

RAM/ROM 関連の制約

- ・ BRAM 使用率 (BRAM_UTILIZATION_RATIO)
- ・ 自動 BRAM パッキング (AUTO_BRAM_PACKING)
- ・ 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)

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/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。アップデートは、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

次のデバイスのブロック RAM リソースでは、異なる読み出し/書き込み同期モードがサポートされます。

- ・ Virtex®-4
- ・ Virtex-5
- ・ Spartan®-3
- ・ Spartan-3E
- ・ Spartan-3A

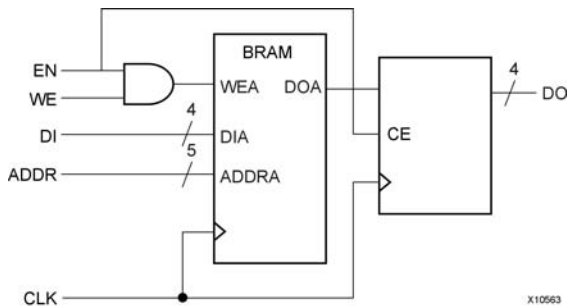
次のコード例では、シングルポートブロックRAMを記述しています。これらの例を応用して、デュアルポートブロックRAMを記述できます。デュアルポートブロックRAMには、各ポートにそれぞれ異なる読み出し/書き込みモードをコンフィギュレーションできます。XSTでは、これらのモードを自動的に推論できます。

次にデバイスによりサポートされる読み出し/書き込みモードと、XSTによる処理方法を示します。

読み出し/書き込みモードのサポート

デバイス	推論モード	ビヘイビア
Spartan-3	WRITE_FIRST	マクロの推論および生成
Spartan-3E	READ_FIRST	NCF ファイル内で生成されたブロックRAMに適切な制約 (WRITE_MODE、WRITE_MODE_A、WRITE_MODE_B) を設定
Spartan-3A	READ_FIRST	
Virtex-4	NO_CHANGE	
Virtex-5	NO_CHANGE	
CPLD	なし	RAM の推論は完全に無効

READ_FIRST モードのシングルポート RAM の図



READ_FIRST モードのシングルポート RAM のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
WE	同期書き込みイネーブル (アクティブ High)
EN	クロック イネーブル
ADDR	読み出し/書き込みアドレス
DI	データ入力
DO	データ出力

READ_FIRST モードのシングル ポート RAM の VHDL コード例 1

```
--
-- Read-First Mode
--
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;
```

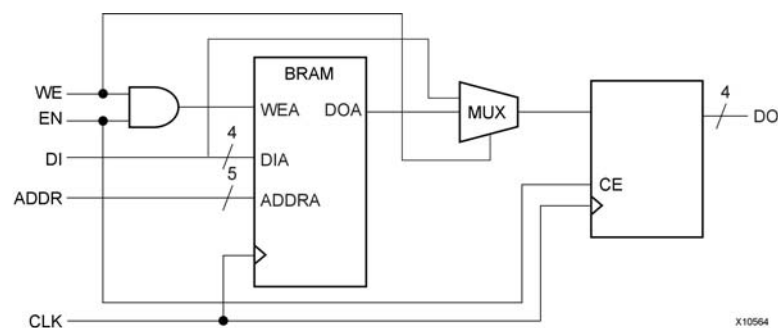
READ_FIRST モードのシングル ポート RAM の Verilog コード例 1

```
//
// Read-First Mode
//
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 RAM の図



WRITE_FIRST モードのシングル ポート RAM のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
WE	同期書き込みイネーブル (アクティブ High)
EN	クロック イネーブル
ADDR	読み出し/書き込みアドレス
DI	データ入力
DO	データ出力

WRITE_FIRST モードのシングル ポート RAM の VHDL コード例 1

```
--
-- Write-First Mode (template 1)
--

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 RAM の VHDL コード例 2

```
--  
-- Write-First Mode (template 2)  
--  
  
library ieee;  
use ieee.std_logic_1164.all;  
use ieee.std_logic_unsigned.all;  
  
entity rams_02b 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_02b;  
  
architecture syn of rams_02b is  
  type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);  
  signal RAM : ram_type;  
  signal read_addr: std_logic_vector(5 downto 0);  
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;  
        read_addr <= addr;  
      end if;  
    end if;  
  end process;  
  
  do <= ram(conv_integer(read_addr));  
  
end syn;
```

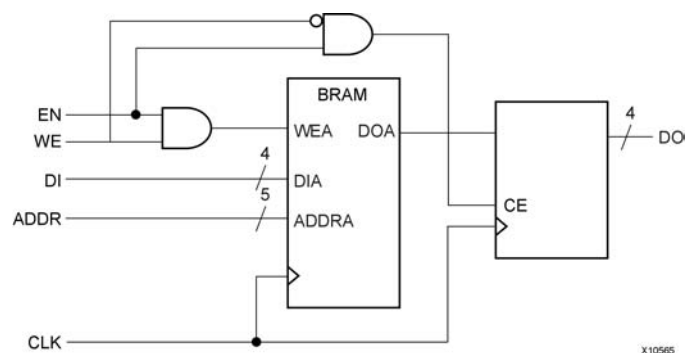

WRITE_FIRST モードのシングル ポート RAM の Verilog コード例 1

```
//  
// Write-First Mode (template 1)  
//  
  
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
```

WRITE_FIRST モードのシングル ポート RAM の Verilog コード例 2

```
//  
// Write-First Mode (template 2)  
//  
  
module v_rams_02b (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    [5:0] read_addr;  
  
    always @(posedge clk)  
    begin  
        if (en)  
        begin  
            if (we)  
            begin  
                RAM[addr] <= di;  
                read_addr <= addr;  
            end  
        end  
  
        assign do = RAM[read_addr];  
    end  
endmodule
```

NO_CHANGE モードのシングル ポート RAM の図



X10565

NO_CHANGE モードのシングル ポート RAM のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
WE	同期書き込みイネーブル (アクティブ High)
EN	クロック イネーブル
ADDR	読み出し/書き込みアドレス
DI	データ入力
DO	データ出力

NO_CHANGE モードのシングル ポート RAM の VHDL コード例 2

```
--
-- No-Change Mode (template 1)
--

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 RAM の Verilog コード例 2

```
//
// No-Change Mode (template 1)
//

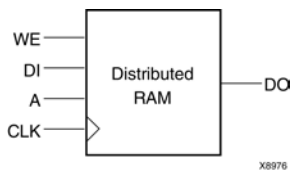
module v_rams_03 (clk, we, en, addr, di, do);

    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
```

次の記述では、分散 RAM のみにインプリメント可能です。

非同期読み出し付きシングル ポート RAM の図



非同期読み出し付きシングル ポート RAM のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
WE	同期書き込みイネーブル (アクティブ High)
A	読み出し/書き込みアドレス
DI	データ入力
DO	データ出力

非同期読み出し付きシングル ポート RAM の VHDL コード例

```
--
-- Single-Port RAM with Asynchronous Read
--

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シングル ポート RAM の Verilog コード例

```
//
// Single-Port RAM with Asynchronous Read
//

module v_rams_04 (clk, we, a, di, do);

    input  clk;
    input  we;
    input  [5:0] a;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] ram [63:0];

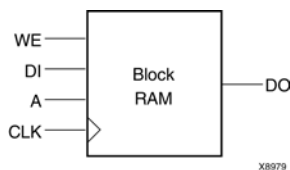
    always @(posedge clk) begin
        if (we)
            ram[a] <= di;
    end

    assign do = ram[a];

endmodule
```

次の記述は、真の同期読み出しをインプリメントします。同期読み出しは Virtex デバイスのブロック RAM の機能で、クロック エッジで読み出しアドレスが格納されます。次の記述は、次の図に示すようにブロック RAM に直接インプリメントできます 同じ記述は、分散 RAM にもマップ可能です。

同期読み出し (透過読み出し) 付きシングル ポート RAM の図



同期読み出し (透過読み出し) 付きシングル ポート RAM のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
WE	同期書き込みイネーブル (アクティブ High)
A	読み出し/書き込みアドレス
DI	データ入力
DO	データ出力

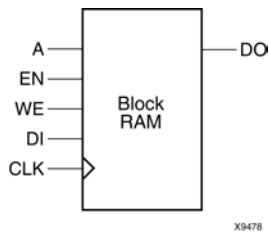
同期読み出し (透過読み出し) 付きシングル ポート RAM の VHDL コード例

```
--  
-- Single-Port RAM with Synchronous Read (Read Through)  
--  
  
library ieee;  
use ieee.std_logic_1164.all;  
use ieee.std_logic_unsigned.all;  
  
entity rams_07 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_07;  
  
architecture syn of rams_07 is  
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);  
    signal RAM : ram_type;  
    signal read_a : std_logic_vector(5 downto 0);  
begin  
  
    process (clk)  
    begin  
        if (clk'event and clk = '1') then  
            if (we = '1') then  
                RAM(conv_integer(a)) <= di;  
            end if;  
            read_a <= a;  
        end if;  
    end process;  
  
    do <= RAM(conv_integer(read_a));  
  
end syn;
```

同期読み出し (透過読み出し) 付きシングル ポート RAM の Verilog コード例

```
//  
// Single-Port RAM with Synchronous Read (Read Through)  
//  
  
module v_rams_07 (clk, we, a, di, do);  
  
    input  clk;  
    input  we;  
    input  [5:0] a;  
    input  [15:0] di;  
    output [15:0] do;  
    reg    [15:0] ram [63:0];  
    reg    [5:0] read_a;  
  
    always @(posedge clk) begin  
        if (we)  
            ram[a] <= di;  
        read_a <= a;  
    end  
  
    assign do = ram[read_a];  
  
endmodule
```

イネーブル付きシングル ポート RAM の図



イネーブル付きシングル ポート RAM のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
EN	グローバル イネーブル
WE	同期書き込みイネーブル (アクティブ High)
A	読み出し/書き込みアドレス
DI	データ入力
DO	データ出力

イネーブル付きシングル ポート RAM の VHDL コード例

```
--
-- Single-Port RAM with Enable
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_08 is
    port (clk : in std_logic;
          en  : 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_08;

architecture syn of rams_08 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
    signal read_a : std_logic_vector(5 downto 0);
begin

    process (clk)
    begin
        if (clk'event and clk = '1') then
            if (en = '1') then
                if (we = '1') then
                    RAM(conv_integer(a)) <= di;
                end if;
                read_a <= a;
            end if;
        end if;
    end process;

    do <= RAM(conv_integer(read_a));

end syn;
```

イネーブル付きシングル ポート RAM の Verilog コード例

```
//
// Single-Port RAM with Enable
//

module v_rams_08 (clk, en, we, a, di, do);

    input  clk;
    input  en;
    input  we;
    input  [5:0] a;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] ram [63:0];
    reg    [5:0] read_a;

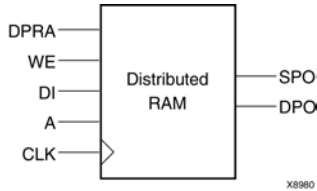
    always @(posedge clk) begin
        if (en)
            begin
                if (we)
                    ram[a] <= di;
                read_a <= a;
            end
        end

        assign do = ram[read_a];
    end

endmodule
```

次の図では、2 つの出力ポートが使用されています。これは、分散 RAM のみに直接マップ可能です。

非同期読み出し付きデュアル ポート RAM の図



非同期読み出し付きデュアル ポート RAM のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
WE	同期書き込みイネーブル (アクティブ High)
A	書き込みアドレス/プライマリ読み出しアドレス
DPRA	デュアル読み出しアドレス
DI	データ入力
SPO	プライマリ出力ポート
DPO	デュアル出力ポート

非同期読み出し付きデュアル ポート RAM の VHDL コード例

```
--
-- Dual-Port RAM with Asynchronous Read
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_09 is
    port (clk      : in std_logic;
          we       : in std_logic;
          a        : in std_logic_vector(5 downto 0);
          dpra     : in std_logic_vector(5 downto 0);
          di       : in std_logic_vector(15 downto 0);
          spo      : out std_logic_vector(15 downto 0);
          dpo      : out std_logic_vector(15 downto 0));
end rams_09;

architecture syn of rams_09 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;

    spo <= RAM(conv_integer(a));
    dpo <= RAM(conv_integer(dpra));

end syn;
```

非同期読み出し付きデュアル ポート RAM の Verilog コード例

```
//
// Dual-Port RAM with Asynchronous Read
//

module v_rams_09 (clk, we, a, dpra, di, spo, dpo);

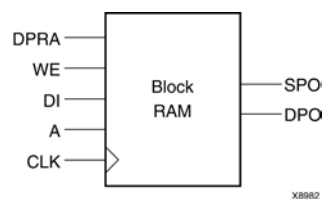
    input  clk;
    input  we;
    input  [5:0] a;
    input  [5:0] dpra;
    input  [15:0] di;
    output [15:0] spo;
    output [15:0] dpo;
    reg    [15:0] ram [63:0];

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

        assign spo = ram[a];
        assign dpo = ram[dpra];
    endmodule
```

次の記述は、次の図に示すようにブロック RAM に直接マップできます 分散 RAM にもインプリメントできます。

同期読み出し (透過読み出し) 付きデュアル ポート RAM の図



同期読み出し (透過読み出し) 付きデュアル ポート RAM のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
WE	同期書き込みイネーブル (アクティブ High)
A	書き込みアドレス/プライマリ読み出しアドレス
DPRA	デュアル読み出しアドレス
DI	データ入力
SPO	プライマリ出力ポート
DPO	デュアル出力ポート

同期読み出し (透過読み出し) 付きデュアル ポート RAM の VHDL コード例

```
--
-- Dual-Port RAM with Synchronous Read (Read Through)
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_1l is
    port (clk      : in std_logic;
          we       : in std_logic;
          a        : in std_logic_vector(5 downto 0);
          dpra     : in std_logic_vector(5 downto 0);
          di       : in std_logic_vector(15 downto 0);
          spo      : out std_logic_vector(15 downto 0);
          dpo      : out std_logic_vector(15 downto 0));
end rams_1l;

architecture syn of rams_1l is
    type ram_type is array (63 downto 0)
        of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
    signal read_a : std_logic_vector(5 downto 0);
    signal read_dpra : std_logic_vector(5 downto 0);
begin

    process (clk)
    begin
        if (clk'event and clk = '1') then
            if (we = '1') then
                RAM(conv_integer(a)) <= di;
            end if;
            read_a <= a;
            read_dpra <= dpra;
        end if;
    end process;

    spo <= RAM(conv_integer(read_a));
    dpo <= RAM(conv_integer(read_dpra));

end syn;
```

同期読み出し (透過読み出し) 付きデュアル ポート RAM の Verilog コード例

```
//
// Dual-Port RAM with Synchronous Read (Read Through)
//

module v_rams_1l (clk, we, a, dpra, di, spo, dpo);

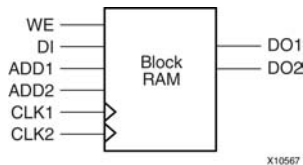
    input  clk;
    input  we;
    input  [5:0] a;
    input  [5:0] dpra;
    input  [15:0] di;
    output [15:0] spo;
    output [15:0] dpo;
    reg    [15:0] ram [63:0];
    reg    [5:0] read_a;
    reg    [5:0] read_dpra;

    always @(posedge clk) begin
        if (we)
            ram[a] <= di;
        read_a <= a;
        read_dpra <= dpra;
    end

    assign spo = ram[read_a];
    assign dpo = ram[read_dpra];

endmodule
```

同期読み出し (透過読み出し) および 2 つのクロック付きデュアル ポート RAM の図



同期読み出し (透過読み出し) および 2 つのクロック付きデュアル ポート RAM のピンの説明

I/O ピン	説明
CLK1	書き込み/プライマリ読み出しクロック (立ち上がりエッジ)
CLK2	デュアル読み出しクロック (立ち上がりエッジ)
WE	同期書き込みイネーブル (アクティブ High)
ADD1	書き込み/プライマリ読み出しアドレス
ADD2	デュアル読み出しアドレス
DI	データ入力
DO1	プライマリ出力ポート
DO2	デュアル出力ポート

同期読み出し (透過読み出し) および 2 つのクロック付きデュアル ポート RAM の VHDL コード例

```
--
-- Dual-Port RAM with Synchronous Read (Read Through)
-- using More than One Clock
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_12 is
    port (clk1 : in std_logic;
          clk2 : in std_logic;
          we   : in std_logic;
          add1 : in std_logic_vector(5 downto 0);
          add2 : in std_logic_vector(5 downto 0);
          di   : in std_logic_vector(15 downto 0);
          do1  : out std_logic_vector(15 downto 0);
          do2  : out std_logic_vector(15 downto 0));
end rams_12;

architecture syn of rams_12 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
    signal read_add1 : std_logic_vector(5 downto 0);
    signal read_add2 : std_logic_vector(5 downto 0);
begin

    process (clk1)
    begin
        if (clk1'event and clk1 = '1') then
            if (we = '1') then
                RAM(conv_integer(add1)) <= di;
            end if;
            read_add1 <= add1;
        end if;
    end process;

    do1 <= RAM(conv_integer(read_add1));

    process (clk2)
    begin
        if (clk2'event and clk2 = '1') then
            read_add2 <= add2;
        end if;
    end process;

    do2 <= RAM(conv_integer(read_add2));

end syn;
```

同期読み出し (透過読み出し) および 2 つのクロック付きデュアル ポート RAM の Verilog コード例

```
//
// Dual-Port RAM with Synchronous Read (Read Through)
// using More than One Clock
//

module v_rams_12 (clk1, clk2, we, add1, add2, di, do1, do2);

    input  clk1;
    input  clk2;
    input  we;
    input  [5:0] add1;
    input  [5:0] add2;
    input  [15:0] di;
    output [15:0] do1;
    output [15:0] do2;
    reg    [15:0] ram [63:0];
    reg    [5:0] read_add1;
    reg    [5:0] read_add2;

    always @(posedge clk1) begin
        if (we)
            ram[add1] <= di;
        read_add1 <= add1;
    end

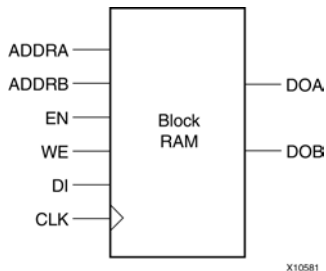
    assign do1 = ram[read_add1];

    always @(posedge clk2) begin
        read_add2 <= add2;
    end

    assign do2 = ram[read_add2];

endmodule
```

1 つのイネーブル信号で両方のポートを制御するデュアル ポート RAM の図



1 つのイネーブル信号で両方のポートを制御するデュアル ポート RAM のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
EN	プライマリ グローバル イネーブル (アクティブ High)
WE	プライマリ同期書き込みイネーブル (アクティブ High)
ADDR A	書き込みアドレス/プライマリ読み出しアドレス
ADDR B	デュアル読み出しアドレス
DI	プライマリ データ入力
DO A	プライマリ出力ポート
DO B	デュアル出力ポート

1 つのイネーブル信号で両方のポートを制御するデュアル ポート RAM の VHDL コード例

```
--
-- Dual-Port RAM with One Enable Controlling Both Ports
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_13 is
    port (clk      : in std_logic;
          en       : in std_logic;
          we       : in std_logic;
          addra    : in std_logic_vector(5 downto 0);
          addrb    : in std_logic_vector(5 downto 0);
          di       : 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_13;

architecture syn of rams_13 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
    signal read_addra : std_logic_vector(5 downto 0);
    signal read_addrb : std_logic_vector(5 downto 0);
begin

    process (clk)
    begin
        if (clk'event and clk = '1') then
            if (en = '1') then
                if (we = '1') then
                    RAM(conv_integer(addra)) <= di;
                end if;

                read_addra <= addra;
                read_addrb <= addrb;

            end if;
        end if;
    end process;

    doa <= RAM(conv_integer(read_addra));
    dob <= RAM(conv_integer(read_addrb));

end syn;
```

1 つのイネーブル信号で両方のポートを制御するデュアル ポート RAM の Verilog コード例

```
//
// Dual-Port RAM with One Enable Controlling Both Ports
//

module v_rams_13 (clk, en, we, addra, addrb, di, doa, dob);

    input  clk;
    input  en;
    input  we;
    input  [5:0] addra;
    input  [5:0] addrb;
    input  [15:0] di;
    output [15:0] doa;
    output [15:0] dob;
    reg    [15:0] ram [63:0];
    reg    [5:0] read_addra;
    reg    [5:0] read_addrb;

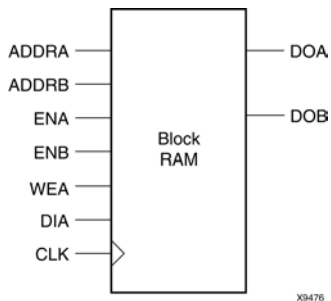
    always @(posedge clk) begin
        if (en)
            begin
                if (we)
                    ram[addra] <= di;
                read_addra <= addra;
                read_addrb <= addrb;
            end
        end

        assign doa = ram[read_addra];
        assign dob = ram[read_addrb];
    end

endmodule
```

次の記述は、次の図に示すようにブロック RAM に直接マップできます

各ポートにそれぞれイネーブル信号があるデュアル ポート RAM の図



各ポートにそれぞれイネーブル信号があるデュアル ポート RAM のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
ENA	プライマリ グローバル イネーブル (アクティブ High)
ENB	デュアル グローバル イネーブル (アクティブ High)
WEA	プライマリ同期書き込みイネーブル (アクティブ High)
ADDRA	書き込みアドレス/プライマリ読み出しアドレス
ADDRB	デュアル読み出しアドレス
DIA	プライマリ データ入力
DOA	プライマリ出力ポート
DOB	デュアル出力ポート

各ポートにそれぞれイネーブル信号があるデュアル ポート RAM の VHDL コード例

```
--
-- Dual-Port RAM with Enable on Each Port
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_14 is
    port (clk      : in std_logic;
          ena      : in std_logic;
          enb      : in std_logic;
          wea      : in std_logic;
          addra    : in std_logic_vector(5 downto 0);
          addrb    : in std_logic_vector(5 downto 0);
          dia      : 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_14;

architecture syn of rams_14 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
    signal read_addra : std_logic_vector(5 downto 0);
    signal read_addrb : std_logic_vector(5 downto 0);
begin

    process (clk)
    begin
        if (clk'event and clk = '1') then

            if (ena = '1') then
                if (wea = '1') then
                    RAM(conv_integer(addra)) <= dia;
                end if;
                read_addra <= addra;
            end if;

            if (enb = '1') then
                read_addrb <= addrb;
            end if;

        end if;
    end process;

    doa <= RAM(conv_integer(read_addra));
    dob <= RAM(conv_integer(read_addrb));

end syn;
```

各ポートにそれぞれイネーブル信号があるデュアル ポート RAM の Verilog コード例

```
//
// Dual-Port RAM with Enable on Each Port
//

module v_rams_14 (clk,ena,enb,wea,addra,addrb,dia,doa,dob);

    input  clk;
    input  ena;
    input  enb;
    input  wea;
    input  [5:0] addra;
    input  [5:0] addrb;
    input  [15:0] dia;
    output [15:0] doa;
    output [15:0] dob;
    reg    [15:0] ram [63:0];
    reg    [5:0] read_addra;
    reg    [5:0] read_addrb;

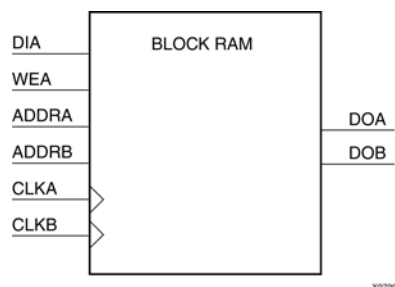
    always @(posedge clk) begin
        if (ena)
            begin
                if (wea)
                    ram[addra] <= dia;
                read_addra <= addra;
            end

        if (enb)
            read_addrb <= addrb;
    end

    assign doa = ram[read_addra];
    assign dob = ram[read_addrb];

endmodule
```

異なるクロックが付いたデュアル ポート ブロック RAM の図

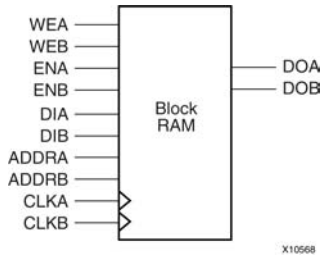


異なるクロックが付いたデュアル ポート ブロック RAM のピンの説明

I/O ピン	説明
CLKA	クロック (立ち上がりエッジ)
CLKB	クロック (立ち上がりエッジ)
WEA	プライマリ同期書き込みイネーブル (アクティブ High)
ADDRA	書き込みアドレス/プライマリ読み出しアドレス
ADDRB	デュアル読み出しアドレス
DIA	プライマリ データ入力
DOA	プライマリ出力ポート
DOB	デュアル出力ポート

書き込みポートが 2 つあるデュアル ポート ブロック RAM は、VHDL と Verilog の両方でサポートされます。デュアル書き込みポートでは、データ ポートが 2 つあるだけでなく、各ポートに個別の書き込みクロックおよび書き込みイネーブルを使用できることがあります。デュアル ポート ブロック RAM には 2 つのクロックがあり、1 つはプライマリの読み出しと書き込みポートで共有され、もう 1 つはセカンダリの読み出しと書き込みポートで共有されるので、各ポートに個別の書き込みクロックを使用する場合、読み出しクロックも個別になることに注意してください。このタイプのブロック RAM は、VHDL では SHARED 変数を使用して記述されています。XST の VHDL アナライザーではこの SHARED 変数が認識されますが、有効な RAM マクロが記述されていないと、HDL 合成段階でエラーが発生します。

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 の図



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 のピンの説明

I/O ピン	説明
CLKA、CLKB	クロック (立ち上がりエッジ)
ENA	プライマリ グローバル イネーブル (アクティブ High)
ENB	デュアル グローバル イネーブル (アクティブ High)
WEA、WEB	プライマリ同期書き込みイネーブル (アクティブ High)
ADDRA	書き込みアドレス/プライマリ読み出しアドレス
ADDRB	デュアル読み出しアドレス
DIA	プライマリ データ入力
DIB:	デュアル データ入力
DOA	プライマリ出力ポート
DOB	デュアル出力ポート

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

次は、一般的なコード例で、異なるクロック、イネーブル、書き込みイネーブルが含まれます。

```
--
-- Dual-Port Block RAM with Two Write Ports
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity rams_16 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(5 downto 0);
         addrb : in std_logic_vector(5 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_16;

architecture syn of rams_16 is
    type ram_type is array (63 downto 0) of std_logic_vector(15 downto 0);
    shared variable RAM : ram_type;
begin

    process (CLKA)
    begin
        if CLKA'event and CLKA = '1' then
            if ENA = '1' then
                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;
```

SHARED 変数があるため、各ポートに対する同期読み出し/書き込みの記述がシングル書き込みポートがある RAM で推奨されるコード例とは異なる場合があります。コードを記述する順序が重要なので注意してください。

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

次は、一般的なコード例で、異なるクロック、イネーブル、書き込みイネーブルが含まれます。

```
//
// Dual-Port Block RAM with Two Write Ports
//

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
```

WRITE_FIRST の同期の VHDL コード例 1

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

WRITE_FIRST の同期の VHDL コード例 2

この例では、書き込みポートの記述の後に読み出しポートを記述する必要があります。

```
process (CLKA)
begin
    if CLKA'event and CLKA = '1' then
        if WEA = '1' then
            RAM(conv_integer(ADDRA)) := DIA;
        end if;
        DOA <= RAM(conv_integer(ADDRA));  -- The read statement must come
                                         -- AFTER the write statement
    end if;
end process;
```

このテンプレートは、次に示すシングル書き込みポート RAM の READ_FIRST の同期のテンプレートと、信号および変数が異なることを除き同じに見えますが、機能が異なります。

```
signal RAM : RAMtype;

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

READ_FIRST の同期のコード例

READ_FIRST の同期を記述する場合は、書き込みポートを記述する前に読み出しポートを記述する必要があります。

```
process (CLKA)
begin
    if CLKA'event and CLKA = '1' then
        DOA <= RAM(conv_integer(ADDRA)); -- The read statement must come
                                         -- BEFORE the write statement

        if WEA = '1' then
            RAM(conv_integer(ADDRA)) := DIA;
        end if;
    end if;
end process;
```

NO_CHANGE の同期のコード例

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

バイト幅の書き込みイネーブルの付いたシングルおよびデュアルポートブロック RAM は、VHDL と Verilog の両方でサポートされます。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、これらの列ごとの書き込みを別々に制御します。

複数の書き込み文を使用する場合、関連する書き込みイネーブルの記述を含む書き込みアクセス文が列ごとに 1 つずつ記述されます。

1 つの書き込み文を使用する場合は、記述できる書き込みアクセス文は 1 つのみになります。書き込みイネーブルは、メインの順次プロセスの外側に記述されます。

列ベースの RAM を記述するこれら 2 つの方法については、次のコード例を参照してください。

XST では、現在のところ 2 つ目のソリューション (1 つの書き込み文の使用) のみがサポートされます。

複数の書き込み文を使用した VHDL コード例

```
type ram_type is array (SIZE-1 downto 0)
    of std_logic_vector (2*WIDTH-1 downto 0);
signal RAM : ram_type;

(...)

process(clk)
begin
    if posedge(clk) then
        if we(1) = '1' then
            RAM(conv_integer(addr))(2*WIDTH-1 downto WIDTH) <= di(2*WIDTH-1 downto WIDTH);
        end if;
        if we(0) = '1' then
            RAM(conv_integer(addr))(WIDTH-1 downto 0) <= di(WIDTH-1 downto 0);
        end if;

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

複数の書き込み文を使用した Verilog コード例

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

always @(posedge clk)
begin
    if (we[1]) then
        RAM[addr][2*WIDTH-1:WIDTH] <= di[2*WIDTH-1:WIDTH];
    end if;
    if (we[0]) then
        RAM[addr][WIDTH-1:0] <= di[WIDTH-1:0];
    end if;

    do <= RAM[addr];
end
```


1 つの書き込み文を使用した VHDL コード例

```

type ram_type is array (SIZE-1 downto 0)
    of std_logic_vector (2*WIDTH-1 downto 0);
signal RAM : ram_type;
signal di0, di1 : std_logic_vector (WIDTH-1 downto 0);

(...)

-- Write enables described outside main sequential process
process(we, di, addr)
begin

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

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

end process;

process(clk)
begin
    if posedge(clk) then
        if en = '1' then
            RAM(conv_integer(addr)) <= di1 & di0; -- single write access statement
            do <= RAM(conv_integer(addr));
        end if;
    end if;
end process;

```

1 つの書き込み文を使用した Verilog コード例

```

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

always @(we or di or addr)
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
    RAM[addr]<={di1,di0};
    do <= RAM[addr];
end

```

次のコード例では、シングルポートブロック RAM を使用して、バイト幅の書き込みイネーブルのテンプレートを簡単に記述していますが、XST ではデュアルポートブロック RAM もサポートされます。

READ_FIRST モード : バイト幅の書き込みイネーブル (2 バイト) 付きシングル ポート BRAM のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
WE	書き込みイネーブル
ADDR	読み出し/書き込みアドレス
DI	データ入力
DO	RAM 出力ポート

READ_FIRST モード : バイト幅の書き込みイネーブル (2 バイト) 付きシングル ポート BRAM の VHDL コード例

```
--
-- Single-Port BRAM with Byte-wide Write Enable (2 bytes) in Read-First Mode
--

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
            RAM(conv_integer(addr)) <= di1 & di0;
            do <= RAM(conv_integer(addr));
        end if;
    end process;
end syn;
```

READ_FIRST モード : バイト幅の書き込みイネーブル (2 バイト) 付きシングル ポート BRAM の Verilog コード例

```
//
// Single-Port BRAM with Byte-wide Write Enable (2 bytes) in Read-First Mode
//

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
        RAM[addr] <= {di1, di0};
        do <= RAM[addr];
    end

endmodule
```

WRITE_FIRST モード : バイト幅の書き込みイネーブル (2 バイト) 付きシングル ポート BRAM のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
WE	書き込みイネーブル
ADDR	読み出し/書き込みアドレス
DI	データ入力
DO	RAM 出力ポート

WRITE_FIRST モード : バイト幅の書き込みイネーブル (2 バイト) 付きシングル ポート BRAM の VHDL コード例

```
--
-- Single-Port BRAM with Byte-wide Write Enable (2 bytes) in Write-First Mode
--

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)
    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;

    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;
```

WRITE_FIRST モード : バイト幅の書き込みイネーブル (2 バイト) 付きシングル ポート BRAM の Verilog コード例

```
//
// Single-Port BRAM with Byte-wide Write Enable (2 bytes) in Write-First Mode
//

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)
    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

            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
        RAM[addr] <= {di1, di0};
        do <= {do1, do0};
    end

endmodule
```

NO_CHANGE モード : バイト幅の書き込みイネーブル (2 バイト) 付きシングル ポート BRAM のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
WE	書き込みイネーブル
ADDR	読み出し/書き込みアドレス
DI	データ入力
DO	RAM 出力ポート

XST では、基本的な 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 BRAM に吸収されます。

NO_CHANGE モード : バイト幅の書き込みイネーブル (2 バイト) 付きシングル ポート BRAM の VHDL コード例

```
--
-- Single-Port BRAM with Byte-wide Write Enable (2 bytes) in No-Change Mode
--

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)
    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;
```

NO_CHANGE モード：バイト幅の書き込みイネーブル (2 バイト) 付きシングル ポート BRAM の Verilog コード例

```
//
// Single-Port BRAM with Byte-wide Write Enable (2 bytes) in No-Change Mode
//

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)
    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

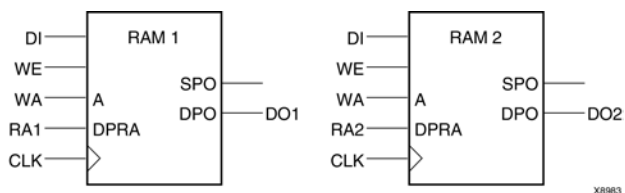
        end

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

endmodule
```

XST では、書き込みアドレスとは異なるアドレスのデータにアクセス可能な読み出しポートが複数ある RAM の記述を認識できます。この場合、使用できる書き込みポートは 1 つのみです。次の記述は、各出力ポートに対する RAM のデータを複製してインプリメントされます (次の図を参照)。

複数ポート RAM の図



複数ポート RAM のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
WE	同期書き込みイネーブル (アクティブ High)
WA	書き込みアドレス
RA1	最初の RAM の読み出しアドレス
RA2	2 番目の RAM の読み出しアドレス
DI	データ入力
DO1	最初の RAM 出力ポート
DO2	2 番目の RAM 出力ポート

複数ポート RAM の VHDL コード例

```
--
-- Multiple-Port RAM Descriptions
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_17 is
  port (clk : in std_logic;
        we  : in std_logic;
        wa  : in std_logic_vector(5 downto 0);
        ra1 : in std_logic_vector(5 downto 0);
        ra2 : in std_logic_vector(5 downto 0);
        di  : in std_logic_vector(15 downto 0);
        do1 : out std_logic_vector(15 downto 0);
        do2 : out std_logic_vector(15 downto 0));
end rams_17;

architecture syn of rams_17 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(wa)) <= di;
      end if;
    end if;
  end process;

  do1 <= RAM(conv_integer(ra1));
  do2 <= RAM(conv_integer(ra2));

end syn;
```

複数ポート RAM の Verilog コード例

```
//
// Multiple-Port RAM Descriptions
//

module v_rams_17 (clk, we, wa, ra1, ra2, di, do1, do2);

    input  clk;
    input  we;
    input  [5:0] wa;
    input  [5:0] ra1;
    input  [5:0] ra2;
    input  [15:0] di;
    output [15:0] do1;
    output [15:0] do2;
    reg    [15:0] ram [63:0];

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

    assign do1 = ram[ra1];
    assign do2 = ram[ra2];

endmodule
```

XST では、Virtex-4、Virtex-5 デバイスおよび関連するブロック RAM リソースで提供されている、データ出力にリセットが付いたブロック RAM がサポートされます。また、RAM データ出力に同期制御の初期化機能を含めることもできます。

リセット可能なデータ ポートは、次の同期モードのブロック RAM に含めることができます。

- ・ リセット付き READ_FIRST モード
- ・ リセット付き WRITE_FIRST モード
- ・ リセット付き NO_CHANGE モード
- ・ レジスタを介したリセット付き ROM
- ・ サポートされているデュアル ポート テンプレート

XST では、デュアル読み出しブロック RAM 記述でデュアル書き込みのブロック RAM がサポートされていないため、両方のデータ出力にリセットを付けることはできませんが、読み出し/書き込みの同期はプライマリ データ出力でのみ可能です。デュアル出力は、READ_FIRST モードのみで使用できます。

リセット付きブロック RAM のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
EN	グローバル イネーブル
WE	書き込みイネーブル (アクティブ High)
ADDR	読み出し/書き込みアドレス
RST	データ出力のリセット
DI	データ入力
DO	RAM 出力ポート

リセット付きブロック RAM の VHDL コード例

```
--
-- Block RAM with Reset
--

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(5 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 (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 -- optional enable

                if we = '1' then -- write enable
                    ram(conv_integer(addr)) <= di;
                end if;

                if rst = '1' then -- optional reset
                    do <= (others => '0');
                else
                    do <= ram(conv_integer(addr)) ;
                end if;

            end if;
        end if;
    end process;

end syn;
```

リセット付きブロック RAM の Verilog コード例

```
//
// Block RAM with Reset
//

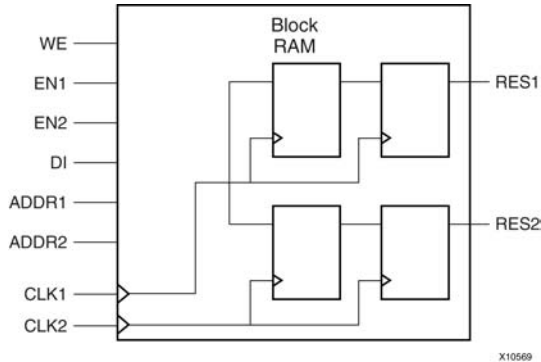
module v_rams_18 (clk, en, we, rst, addr, di, do);

    input  clk;
    input  en;
    input  we;
    input  rst;
    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) // optional enable
        begin
            if (we) // write enable
                ram[addr] <= di;

            if (rst) // optional reset
                do <= 16'h0000;
            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 の図



オプションの出力レジスタ付きブロック RAM のピンの説明

I/O ピン	説明
CLK1、CLK2	クロック (立ち上がりエッジ)
WE	書き込みイネーブル
EN1、EN2	クロック イネーブル (アクティブ High)
ADDR1	プライマリ読み出しアドレス
ADDR2	デュアル読み出しアドレス
DI	データ入力
RES1	プライマリ出力ポート
RES2	デュアル出力ポート

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
--

library IEEE;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity rams_19 is
    port (clk1, clk2 : in std_logic;
          we, en1, en2 : in std_logic;
          addr1 : in std_logic_vector(5 downto 0);
          addr2 : in std_logic_vector(5 downto 0);
          di : in std_logic_vector(15 downto 0);
          res1 : out std_logic_vector(15 downto 0);
          res2 : out std_logic_vector(15 downto 0));
end rams_19;

architecture beh of rams_19 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal ram : ram_type;
    signal do1 : std_logic_vector(15 downto 0);
    signal do2 : std_logic_vector(15 downto 0);
begin

    process (clk1)
    begin
        if rising_edge(clk1) then
            if we = '1' then
                ram(conv_integer(addr1)) <= di;
            end if;
            do1 <= ram(conv_integer(addr1));
        end if;
    end process;

    process (clk2)
    begin
        if rising_edge(clk2) then
            do2 <= ram(conv_integer(addr2));
        end if;
    end process;

    process (clk1)
    begin
        if rising_edge(clk1) then
            if en1 = '1' then
                res1 <= do1;
            end if;
        end if;
    end process;

    process (clk2)
    begin
        if rising_edge(clk2) then
            if en2 = '1' then
                res2 <= do2;
            end if;
        end if;
    end process;

end beh;
```

オプションの出力レジスタ付き ブロック RAM の Verilog コード例

```
//  
// Block RAM with Optional Output Registers  
//  
module v_rams_19 (clk1, clk2, we, en1, en2, addr1, addr2, di, res1, res2);  
  
    input  clk1;  
    input  clk2;  
    input  we, en1, en2;  
    input  [5:0] addr1;  
    input  [5:0] 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 [63:0];  
    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
```

RAM の初期化のコード例

コード例は、本書が作成された時点のものです。アップデートは、
ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

ブロック RAM および分散 RAM の初期内容は、HDL コードでメモリ アレイを記述した信号を初期化すると指定できます。初期化は、VHDL コードで直接することもできますが、初期化データが含まれるファイルを指定することでもできます。詳細は、次を参照してください。

- ・ RAM を HDL コードで直接初期化する例
- ・ 外部ファイルからの RAM の初期化

RAM の初期化は、VHDL および Verilog の両方でサポートされます。

RAM を HDL コードで直接初期化する例

この例では、RAM を Hardware Description Language (HDL) コードで直接初期化する例を示しています。その他詳細は、この後の「[外部ファイルからの RAM の初期化](#)」を参照してください。

RAM の初期内容の VHDL コード例 (16 進数)

RAM の初期内容は、次の例のように VHDL コードでメモリ アレイを記述する信号を初期化することで指定します。

```
...
type ram_type is array (0 to 63) 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"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");
...
process (clk)
begin
  if rising_edge(clk) then
    if we = '1' then
      RAM(conv_integer(a)) <= di;
    end if;
    ra <= a;
  end if;
end process;
...
do <= RAM(conv_integer(ra));
```

ブロック RAM を初期化する Verilog コード例 (16 進数)

RAM の初期内容は、次の例のように initial 文を使用して Verilog コードでメモリ アレイを記述する信号を初期化することで指定します。

```
...
reg [19:0] ram [63:0];
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[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
```

RAM の初期内容の VHDL コード例 (2 進数)

RAM の初期値は、「RAM の初期内容の VHDL コード例 (16 進数)」のように 16 進数で指定するか、次の例のように 2 進数で指定できます。

```
...
type ram_type is array (0 to SIZE-1) of std_logic_vector(15 downto 0);
signal RAM : ram_type :=
(
  "0111100100000101",
  "0000010110111101",
  "1100001101010000",
  ...
  "0000100101110011");
```

ブロック RAM を初期化する Verilog コード例 (2 進数)

RAM の初期値は、「ブロック RAM を初期化する Verilog コード例 (16 進数)」のように 16 進数で指定するか、次の例のように 2 進数で指定できます。

```
...
reg [15:0] ram [63:0];
initial begin
    ram[63] = 16'b0111100100000101;
    ram[62] = 16'b0000010110111101;
    ram[61] = 16'b1100001101010000;
    ...
    ram[0] = 16'b0000100101110011;
end
...
```

シングルポート ブロック RAM の初期内容を記述した VHDL コード例

```
--
-- Initializing Block RAM (Single-Port BRAM)
--
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"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");

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;
```


シングル ポート ブロック RAM の初期内容を記述した Verilog コード例

```
//  
// Initializing Block RAM (Single-Port BRAM)  
//  
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  
  
endmodule
```

デュアル ポート ブロック RAM の初期内容を記述した VHDL コード例

```
--
-- Initializing Block RAM (Dual-Port BRAM)
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_20b is
  port (clk1 : in std_logic;
        clk2 : in std_logic;
        we : in std_logic;
        addr1 : in std_logic_vector(7 downto 0);
        addr2 : in std_logic_vector(7 downto 0);
        di : in std_logic_vector(15 downto 0);
        do1 : out std_logic_vector(15 downto 0);
        do2 : out std_logic_vector(15 downto 0));
end rams_20b;

architecture syn of rams_20b is

  type ram_type is array (255 downto 0) of std_logic_vector (15 downto 0);
  signal RAM : ram_type:= (255 downto 100 => X"B8B8", 99 downto 0 => X"8282");

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;

end syn;
```

デュアル ポート ブロック RAM の初期内容を記述した Verilog コード例

```
//
// Initializing Block RAM (Dual-Port BRAM)
//

module v_rams_20b (clk1, clk2, we, addr1, addr2, di, do1, do2);
  input clk1, clk2;
  input we;
  input [7:0] addr1, addr2;
  input [15:0] di;
  output [15:0] do1, do2;

  reg [15:0] ram [255:0];
  reg [15:0] do1, do2;
  integer index;

  initial begin
    for (index = 0 ; index <= 99 ; index = index + 1) begin
      ram[index] = 16'h8282;
    end

    for (index= 100 ; index <= 255 ; index = index + 1) begin
      ram[index] = 16'hB8B8;
    end
  end

  always @(posedge clk1)
  begin
    if (we)
      ram[addr1] <= di;
    do1 <= ram[addr1];
  end

  always @(posedge clk2)
  begin
    do2 <= ram[addr2];
  end
endmodule
```

外部ファイルからの RAM の初期化

この例では、外部データファイルを使用してブロック RAM を初期化しています。その他詳細は、前述の「RAM を Hardware Description Language (HDL) コードで直接初期化する例」を参照してください。

外部ファイルに含まれる値で RAM を初期化するには、VHDL コードで read 関数を使用します。詳細は、「[サポートされる VHDL ファイル タイプ](#)」を参照してください。次の手順に従って、初期化ファイルを設定します。

- ・ RAM の指定した行の初期内容を表すには、初期化ファイルの各ラインを使用します。
- ・ RAM の内容は 16 進数または 2 進数で表現します。
- ・ RAM のアレイの行数と初期化ファイルの行数を同数にします。
- ・ 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。

```
000011111000011110000111100001111
01001010001000001100000010000100
00000000001111100000000001000001
11111101010000011100010000100100
00001111000011110000111100001111
01001010001000001100000010000100
00000000001111100000000001000001
11111101010000011100010000100100
```

ブロック RAM の初期化 (外部データ ファイル)

RAM の初期値は、外部データ ファイルに保存し、HDL コード内で指定して読み込むことができます。データ ファイルは、コメントまたはその他の情報が何も無い状態の 2 進数または 16 進数で記述されます。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。どちらの例も、参照されたデータ ファイルは rams_20c.data です。

```
00001111000011110000111100001111
01001010001000001100000010000100
000000000011111000000000001000001
11111101010000011100010000100100
00001111000011110000111100001111
01001010001000001100000010000100
000000000011111000000000001000001
11111101010000011100010000100100
```

ブロック RAM の初期化 (外部データ ファイル) の VHDL コード例

次の例では、初期値を生成するループが RAM のアドレス範囲内にあることを確認することで制御されます。この例では、外部データ ファイルを使用してブロック RAM を初期化しています。

```
--
-- Initializing Block RAM from external data file
--

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;
      dout <= to_stdlogicvector(RAM(conv_integer(addr)));
    end if;
  end process;

end syn;
```

外部データ ファイルに含まれる行数が不十分の場合、次のエラー メッセージが表示されます。
ERROR:Xst - raminitfile1.vhd line 40: Line <RamFileLine has not enough elements for target <RAM<63>>.

ブロック RAM の初期化 (外部データ ファイル) の Verilog コード例

外部ファイルに含まれる値で RAM を初期化するには、Verilog コードで \$readmemb または \$readmemh システム タスクを使用します。詳細は、「[Verilog ビヘイビア記述のサポート](#)」を参照してください。次の手順に従って、初期化ファイルを設定します。

- ・ 初期化ファイルの各行が RAM の該当する行の初期内容を記述するようにします。
- ・ RAM の内容は 16 進数または 2 進数で表現します。
- ・ 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 タスクでは次の例に示す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);
```

RAM のアレイの行数と初期化ファイルの行数を同じにする必要があります。

```
//  
// Initializing Block RAM from external data file  
//  
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);  
    end  
  
    always @(posedge clk)  
    begin  
        if (we)  
            ram[addr] <= din;  
            dout <= ram[addr];  
    end  
  
endmodule
```

ブロック RAM リソースを使用した ROM の HDL コーディング手法

XST では、ブロック RAM リソースを使用して同期出力またはアドレス入力を持つ ROM をインプリメントできます。こういった ROM は、HDL 記述によって、シングル ポートまたはデュアル ポートのブロック RAM としてインプリメントされます。

[階層の維持 \(KEEP_HIERARCHY\)](#) を no に設定すると、階層が違っていてもブロック ROM が推論されます。この場合、ROM とデータ出力、またはアドレス レジスタは別の階層ブロックに記述できます。これは、アドバンス HDL 合成段階で推論されます。

ブロック RAM リソースを使用して ROM をインプリメントする場合は、[ROM スタイル \(ROM_STYLE\)](#) 制約を使用して制御します。[ROM スタイル \(ROM_STYLE\)](#) の詳細は、「[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、「[FPGA の最適化](#)」を参照してください。

ブロック RAM リソースを使用した ROM のログ ファイル

```

=====
*                               HDL Synthesis                               *
=====
Synthesizing Unit <rams_21a>.
  Related source file is "rams_21a.vhd".
  Found 64x20-bit ROM for signal <$varindex0000> created at line 38.
  Found 20-bit register for signal <data>.
  Summary:
    inferred   1 ROM(s).
    inferred  20 D-type flip-flop(s).
Unit <rams_21a> synthesized.
=====
HDL Synthesis Report
Macro Statistics
# ROMs                                     : 1
  64x20-bit ROM                           : 1
# Registers                               : 1
  20-bit register                         : 1
=====
*                               Advanced HDL Synthesis                       *
=====
INFO:Xst - Unit <rams_21a> : The ROM <Mrom__varindex0000> will be implemented
as a read-only BLOCK RAM, absorbing the register: <data>.

-----
| ram_type           | Block                                     |      |
-----
| Port A             |
|   aspect ratio     | 64-word x 20-bit (6.9%)                 |      |
|   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>                 |      |
-----
=====
Advanced HDL Synthesis Report
Macro Statistics
# RAMs                                     : 1
  64x20-bit single-port block RAM         : 1
=====

```

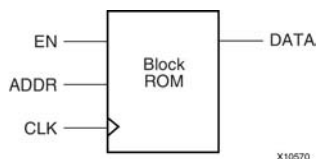
ブロック RAM リソースを使用した ROM 関連の制約

ROM スタイル (ROM_STYLE)

ブロック RAM リソースを使用した 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。アップデートは、
ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

レジスタ付き出力を持つ ROM の図



レジスタ付き出力を持つ ROM のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
EN	同期イネーブル (アクティブ High)
ADDR	読み出しアドレス
DATA	データ出力

レジスタ付き出力を持つ ROM の VHDL コード例 1

```
--
-- ROMs Using Block RAM Resources.
-- VHDL code for a ROM with registered output (template 1)
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_21a is
    port (clk : in std_logic;
          en  : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          data : out std_logic_vector(19 downto 0));
end rams_21a;

architecture syn of rams_21a is

    type rom_type is array (63 downto 0) of std_logic_vector (19 downto 0);
    signal ROM : rom_type:= (X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A",
                             X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
                             X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
                             X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
                             X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
                             X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
                             X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
                             X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
                             X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
                             X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
                             X"0030D", X"02341", X"08201", X"0400D");

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;
```


レジスタ付き出力を持つ ROM の VHDL コード例 2

```
--
-- ROMs Using Block RAM Resources.
-- VHDL code for a ROM with registered output (template 2)
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_21b is
port (clk : in std_logic;
      en : in std_logic;
      addr : in std_logic_vector(5 downto 0);
      data : out std_logic_vector(19 downto 0));
end rams_21b;

architecture syn of rams_21b is
    type rom_type is array (63 downto 0) of std_logic_vector (19 downto 0);
    signal ROM : rom_type:= (X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A",
                             X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
                             X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
                             X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
                             X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
                             X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
                             X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
                             X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
                             X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
                             X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
                             X"0030D", X"02341", X"08201", X"0400D");

    signal rdata : std_logic_vector(19 downto 0);
begin
    rdata <= ROM(conv_integer(addr));

    process (clk)
    begin
        if (clk'event and clk = '1') then
            if (en = '1') then
                data <= rdata;
            end if;
        end if;
    end process;
end syn;
```

レジスタ付き出力を持つ ROM の Verilog コード例 1

```
//  
// ROMs Using Block RAM Resources.  
// Verilog code for a ROM with registered output (template 1)  
//  
  
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  
    end  
endmodule
```

レジスタ付き出力を持つ ROM の Verilog コード例 2

```
//
// ROMs Using Block RAM Resources.
// Verilog code for a ROM with registered output (template 2)
//

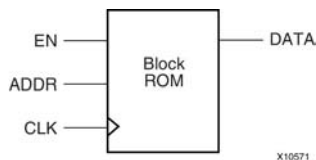
module v_rams_21b (clk, en, addr, data);

    input      clk;
    input      en;
    input      [5:0] addr;
    output reg [19:0] data;
    reg        [19:0] rdata;

    always @(addr) begin
        case(addr)
            6'b000000: rdata <= 20'h0200A; 6'b100000: rdata <= 20'h02222;
            6'b000001: rdata <= 20'h00300; 6'b100001: rdata <= 20'h04001;
            6'b000010: rdata <= 20'h08101; 6'b100010: rdata <= 20'h00342;
            6'b000011: rdata <= 20'h04000; 6'b100011: rdata <= 20'h0232B;
            6'b000100: rdata <= 20'h08601; 6'b100100: rdata <= 20'h00900;
            6'b000101: rdata <= 20'h0233A; 6'b100101: rdata <= 20'h00302;
            6'b000110: rdata <= 20'h00300; 6'b100110: rdata <= 20'h00102;
            6'b000111: rdata <= 20'h08602; 6'b100111: rdata <= 20'h04002;
            6'b001000: rdata <= 20'h02310; 6'b101000: rdata <= 20'h00900;
            6'b001001: rdata <= 20'h0203B; 6'b101001: rdata <= 20'h08201;
            6'b001010: rdata <= 20'h08300; 6'b101010: rdata <= 20'h02023;
            6'b001011: rdata <= 20'h04002; 6'b101011: rdata <= 20'h00303;
            6'b001100: rdata <= 20'h08201; 6'b101100: rdata <= 20'h02433;
            6'b001101: rdata <= 20'h00500; 6'b101101: rdata <= 20'h00301;
            6'b001110: rdata <= 20'h04001; 6'b101110: rdata <= 20'h04004;
            6'b001111: rdata <= 20'h02500; 6'b101111: rdata <= 20'h00301;
            6'b010000: rdata <= 20'h00340; 6'b110000: rdata <= 20'h00102;
            6'b010001: rdata <= 20'h00241; 6'b110001: rdata <= 20'h02137;
            6'b010010: rdata <= 20'h04002; 6'b110010: rdata <= 20'h02036;
            6'b010011: rdata <= 20'h08300; 6'b110011: rdata <= 20'h00301;
            6'b010100: rdata <= 20'h08201; 6'b110100: rdata <= 20'h00102;
            6'b010101: rdata <= 20'h00500; 6'b110101: rdata <= 20'h02237;
            6'b010110: rdata <= 20'h08101; 6'b110110: rdata <= 20'h04004;
            6'b010111: rdata <= 20'h00602; 6'b110111: rdata <= 20'h00304;
            6'b011000: rdata <= 20'h04003; 6'b111000: rdata <= 20'h04040;
            6'b011001: rdata <= 20'h0241E; 6'b111001: rdata <= 20'h02500;
            6'b011010: rdata <= 20'h00301; 6'b111010: rdata <= 20'h02500;
            6'b011011: rdata <= 20'h00102; 6'b111011: rdata <= 20'h02500;
            6'b011100: rdata <= 20'h02122; 6'b111100: rdata <= 20'h0030D;
            6'b011101: rdata <= 20'h02021; 6'b111101: rdata <= 20'h02341;
            6'b011110: rdata <= 20'h00301; 6'b111110: rdata <= 20'h08201;
            6'b011111: rdata <= 20'h00102; 6'b111111: rdata <= 20'h0400D;
        endcase
    end

    always @(posedge clk) begin
        if (en)
            data <= rdata;
    end
endmodule
```

レジスタ付きアドレス入力を持つ ROM の図



レジスタ付きアドレス入力を持つ ROM のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
EN	同期イネーブル (アクティブ High)
ADDR	読み出しアドレス
DATA	データ出力
CLK	クロック (立ち上がりエッジ)

レジスタ付きアドレス入力を持つ ROM の VHDL コード例

```
--
-- ROMs Using Block RAM Resources.
-- VHDL code for a ROM with registered address
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_2lc is
port (clk : in std_logic;
      en : in std_logic;
      addr : in std_logic_vector(5 downto 0);
      data : out std_logic_vector(19 downto 0));
end rams_2lc;

architecture syn of rams_2lc is
    type rom_type is array (63 downto 0) of std_logic_vector (19 downto 0);
    signal ROM : rom_type:= (X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A",
                             X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
                             X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
                             X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
                             X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
                             X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
                             X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
                             X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
                             X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
                             X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
                             X"0030D", X"02341", X"08201", X"0400D");

    signal raddr : std_logic_vector(5 downto 0);
begin
    process (clk)
    begin
        if (clk'event and clk = '1') then
            if (en = '1') then
                raddr <= addr;
            end if;
        end if;
    end process;

    data <= ROM(conv_integer(raddr));
end syn;
```

レジスタ付きアドレス入力を持つ ROM の Verilog コード例

```
//
// ROMs Using Block RAM Resources.
// Verilog code for a ROM with registered address
//

module v_rams_21c (clk, en, addr, data);

    input      clk;
    input      en;
    input      [5:0] addr;
    output reg [19:0] data;
    reg        [5:0] raddr;

    always @(posedge clk) begin
        if (en)
            raddr <= addr;
    end

    always @(raddr) begin
        case(raddr)
            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
```

パイプライン化された分散 RAM の HDL コーディング手法

XST でパイプライン化された分散 RAM を推論させると、デザイン スピードを向上できます。パイプライン化すると、分散 RAM の各段の間にレジスタを挿入することにより、デザインの全体の周波数を大幅に向上することができます。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](#)」で説明されている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 コードで必要なレジスタを記述します。
2. それらのレジスタを分散 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 では最大の分散 RAM の速度に到達させるため、次の両方の場合に使用可能なレジスタの最大数を使用します。

- ・ パイプラインに有効なレジスタが検出される場合
- ・ RAM_STYLE が **pipe_distributed** に設定される場合

XST では、各 RAM で周波数を最大にするために使用するレジスタの最大数が自動的に計算されます。

アドバンス HDL 合成段階中、XST HDL Advisor からは次の場合に最適なレジスタ ステージ数を指定するようにメッセージが表示されます。

- ・ まだ十分な数のレジスタ ステージを指定していない場合
- ・ RAM_STYLE が信号に直接コード記述されている場合

XST では、次の場合に未使用のステージがシフトレジスタとしてインプリメントされます。

- ・ 乗算器の後に配置されたレジスタの数が必要な最大数を超える場合
- ・ シフトレジスタの抽出がオンになっている場合

レジスタに非同期セット/リセット信号が含まれていると、RAM を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。

パイプライン化された分散 RAM のログ ファイル

次は、パイプライン化された分散 RAM のログ ファイルです。

```
=====
*                               HDL Synthesis                               *
=====
Synthesizing Unit <rams_22>.
  Related source file is "rams_22.vhd".
  Found 64x4-bit single-port RAM for signal <RAM>.
  Found 4-bit register for signal <do>.
  Summary:
    inferred   1 RAM(s).
    inferred   4 D-type flip-flop(s).
Unit <rams_22> synthesized.

=====
HDL Synthesis Report

Macro Statistics
# RAMs                                : 1
  64x4-bit single-port RAM            : 1
# Registers                           : 1
  4-bit register                      : 1

=====
*                               Advanced HDL Synthesis                       *
=====
INFO:Xst - Unit <rams_22> : The RAM <Mram_RAM> will be implemented as a
distributed RAM, absorbing the following register(s): <do>.

-----
| aspect ratio | 64-word x 4-bit | | |
| clock        | connected to signal <clk> | | rise |
| write enable | connected to signal <we> | | high |
| address      | connected to signal <addr> | | |
| data in      | connected to signal <di> | | |
| data out     | connected to internal node | | |
| ram_style    | distributed      | | |
-----

Synthesizing (advanced) Unit <rams_22>.
Found pipelined ram on signal <_varindex0000>:
- 1 pipeline level(s) found in a register on signal <_varindex0000>.
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 _varindex0000.
Unit <rams_22> synthesized (advanced).
=====
Advanced HDL Synthesis Report
Macro Statistics
# RAMs                                : 1
  64x4-bit registered single-port distributed RAM : 1
=====
```

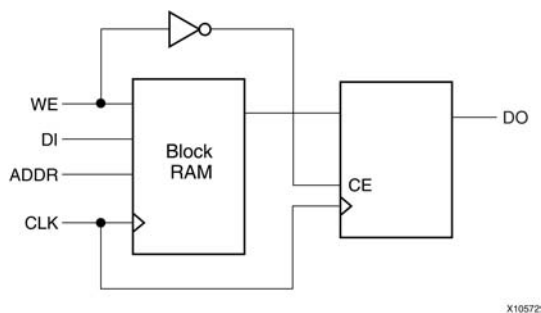
パイプライン化された分散 RAM 関連の制約

- ・ 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)

パイプライン化された分散 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。アップデートは、
ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

パイプライン化された分散 RAM の図



パイプライン化された分散 RAM のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
WE	同期書き込みイネーブル (アクティブ High)
ADDR	読み出し/書き込みアドレス
DI	データ入力
DO	データ出力

パイプライン化された分散 RAM の VHDL コード例

```
--
-- Pipeline distributed RAMs
--

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 RAMs
//

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

endmodule
```

Finite State Machine (FSM) の HDL コーディング手法

Xilinx Synthesis Technology (XST) では、次が実行できます。

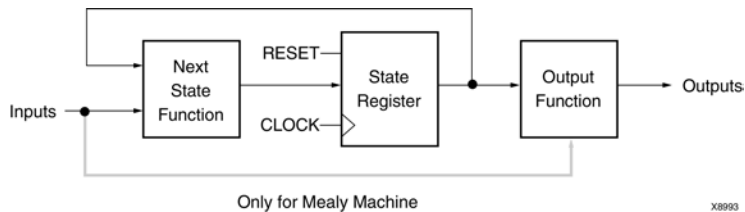
- ・ Finite State Machine (FSM) には、多数のテンプレートがあります。
- ・ 複数のステート エンコード手法を適用してパフォーマンスを向上またはエリアを削減可能
- ・ ユーザーの最初のエンコードを再度エンコード可能
- ・ 同期ステート マシンのみを処理可能

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。

Finite State Machine (FSM) コンポーネントの記述

Finite State Machine (FSM) を記述する方法は多数あります。従来からの方法では、次の図に示すように、ミーリ マシンまたはムーア マシンが使用されます。XST では、両方ともサポートされます。

ミーリ マシンおよびムーア マシンを取り入れた Finite State Machine (FSM) の図



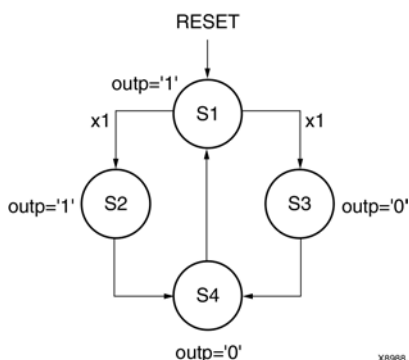
HDL では、FSM コンポーネントの記述に process ブロック (VHDL) および always ブロック (Verilog) を使用するのが最適です。ここでの説明では、「プロセス」という言葉で VHDL の process ブロックと Verilog の always ブロックの両方を示します。

モデルの異なる部分をどのように分割するかによって、1 つの記述に複数のプロセス (1、2、または 3) を含めることができます。次は、非同期リセット (RESET) 付きのムーア マシンの例です。

- ・ 4 つのステート : s1、s2、s3、s4
- ・ 5 つの遷移
- ・ 1 つの入力 : **x1**
- ・ 1 つの出力 : **OUTP**

このモデルは、次のステート ダイアグラムで表すことができます。

ステート ダイアグラム



ステート レジスタ

ステートレジスタは、非同期または同期信号で初期化するか、[レジスタ電源投入時の値 \(REGISTER_POWERUP\)](#) 制約でパワー アップ値を定義しておかないと認識されません。非同期および同期の初期化信号を記述する方法については「[レジスタの HDL コード手法](#)」にあるコード例を参照してください。

VHDL の場合、ステートレジスタは次のような異なるタイプにできます。

- ・ `integer`
- ・ `bit_vector`
- ・ `std_logic_vector`

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便利です。

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を使用できますが、次のようにステートを割り当てることもできます。

```
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。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。

unreachable ステート

XST では、FSM 内の unreachable ステートを検出できます。検出されたステートは、HDL 合成段階でログ ファイルに記述されます。

Finite State Machine (FSM)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、順次プロセスで代入する必要があります。

Finite State Machine (FSM)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 では、次のステート エンコード手法がサポートされます。

- ・ 自動ステート エンコード
- ・ ワンホット ステート エンコード
- ・ グレイ ステート エンコード
- ・ コンパクト ステート エンコード
- ・ ジョンソン ステート エンコード
- ・ シーケンシャル ステート エンコード
- ・ Speed1 ステート エンコード
- ・ ユーザー ステート エンコード

自動ステート エンコード

自動ステート エンコードでは、XST により各 FSM に最適なエンコード アルゴリズムが選択されます。

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、各ステートに 1 つのコード ビットおよび 1 つのフリップフロップを割り当てます。1 つのクロック サイクルで 1 つのステート変数のみがアサートになります。あるステートから別のステートに遷移するときには、2 つのステートのみが切り替わります。ほとんどの FPGA デバイスの場合、フリップフロップが多数があるため、ワンホット エンコードが適しています。スピードを最適化する場合や、消費電力を低減する場合にも適した手法です。

グレイ ステート エンコード

グレイ ステート エンコードでは、あるステートから別のステートに遷移する際、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。また、この手法ではハザードやグリッチを最小限に抑えることができ、T フリップフロップを含むステートレジスタのインプリメンテーションで良い結果が得られます。

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、ステートレジスタに列挙型を使用する場合、[列挙型エンコード手法 \(ENUM_ENCODING\)](#) 制約を使用して各ステートに特定の 2 進数値を割り当てることができます。詳細は、「[デザイン制約](#)」を参照してください。

RAM ベースの Finite State Machine (FSM) 合成

大型の Finite State Machine (FSM) コンポーネントは、Virtex® 以降のデバイスに搭載されているブロック RAM リソースにインプリメントすると、コンパクトで高速にできます。[FSM スタイル \(FSM_STYLE\)](#) を使用すると、FSM にブロック RAM リソースが使用されます。

[FSM スタイル \(FSM_STYLE\)](#) の値には、次があります。

- ・ **lut** (デフォルト)
XST では、LUT を使用して FSM をマップします。
- ・ **bram**
XST では、ブロック RAM 上に FSM をマップします。

[FSM スタイル \(FSM_STYLE\)](#) は次のように指定します。

- ・ ISE® Design Suite
[Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスで、[HDL Options] ページにある [FSM Style] プロパティを [LUT] または [Bram] に設定します。
- ・ コマンド ライン
-fsm_style オプションを使用します。
- ・ HDL コード
[FSM スタイル \(FSM_STYLE\)](#) を使用します。

ブロック RAM にステート マシンをインプリメントできない場合は、次のように処理されます。

- ・ ログ ファイルのアドバンス HDL 合成部分に警告が表示されます。
- ・ ステート マシンが自動的に LUT を使用してインプリメントされます。

たとえば、FSM に非同期リセットがある場合、ブロック RAM にはインプリメントできません。その場合、次のようなメッセージが表示されます。

```
...
=====
*                               Advanced HDL Synthesis                               *
=====

WARNING:Xst - Unable to fit FSM <FSM_0> in BRAM (reset is asynchronous).
Selecting encoding for FSM_0 ...
Optimizing FSM <FSM_0> on signal <current_state> with one-hot encoding.
...
```

セーフ Finite State Machine (FSM) インプリメンテーション

XST では、ステート マシンが不正なステートから回復できるようにするロジックを Finite State Machine (FSM) のインプリメンテーションに追加できます。ステート マシンが実行中に不正の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すことができます。これは、セーフ インプリメンテーション モードと呼ばれます。

FSM のセーフ インプリメンテーションの設定方法 :

- ・ ISE® Design Suite から [Synthesize -XST] プロセスの [Process Properties] ダイアログボックスを表示して、[HDL Options] ページで [Safe Implementation] オプションを [Yes] にします。
- ・ ステートレジスタを表現する階層ブロックまたは信号に [セーフ インプリメンテーション \(SAFE_IMPLEMENTATION\)](#) 制約を設定します。

XST では、デフォルトでリセットステートがリカバリステートとして自動的に選択されます。FSM に初期化信号がない場合は、パワーアップステートがリカバリステートとして選択されます。 [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。

Finite State Machine (FSM) ログ ファイル

XST ログファイルには、マクロの認識段階で認識された Finite State Machine (FSM) の情報がすべて示されます。FSM のエンコード アルゴリズムが自動的に選択されるよう設定している場合は、選択されたアルゴリズムが示されます。

エンコード手法が選択されると、FSM の元のエンコードと FSM エンコードがレポートされます。使用するデバイス ファミリが FPGA の場合、エンコード手法は HDL 合成段階でレポートされます。CPLD の場合は、下位レベルの最適化段階でレポートされます。

```
...
Synthesizing Unit <fsm_1>.
Related source file is "/state_machines_1.vhd".
Found finite state machine <FSM_0> for signal <state>.
-----
| States           | 4 |
| Transitions      | 5 |
| Inputs           | 1 |
| Outputs          | 4 |
| Clock            | clk (rising_edge) |
| Reset            | reset (positive) |
| Reset type       | asynchronous |
| Reset State      | s1 |
| Power Up State   | s1 |
| Encoding         | automatic |
| Implementation   | LUT |
-----
Found 1-bit register for signal <outp>.
Summary:
    inferred 1 Finite State Machine(s).
    inferred 1 D-type flip-flop(s).
Unit <fsm_1> synthesized.

=====
HDL Synthesis Report

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

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

Advanced Registered AddSub inference ...
Analyzing FSM <FSM_0> for best encoding.
Optimizing FSM <state/FSM_0> on signal <state[1:2]> with gray encoding.
-----
State | Encoding
-----
s1    | 00
s2    | 01
s3    | 11
s4    | 10
-----
=====
HDL Synthesis Report

Macro Statistics
# FSMs                : 1
=====
```

Finite State Machine (FSM) 関連の制約

- ・ Finite State Machine (FSM) 自動抽出 (FSM_EXTRACT)
- ・ Finite State Machine (FSM) スタイル (FSM_STYLE)
- ・ Finite State Machine (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)

Finite State Machine (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。アップデートは、
ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

1 つのプロセス ブロックを使用した Finite State Machine (FSM) のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
RESET	非同期リセット (アクティブ High)
X1	FSM の入力
OUTP	FSM の出力

1 つのプロセス ブロックを使用した Finite State Machine (FSM) の VHDL コード例

```
--
-- State Machine with a single process.
--

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 beh1 of fsm_1 is
    type state_type is (s1,s2,s3,s4);
    signal state: state_type ;
begin

    process (clk,reset)
    begin
        if (reset = '1') then
            state <= s1;
            outp <= '1';
        elsif (clk='1' and clk'event) then
            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 process;

end beh1;
```

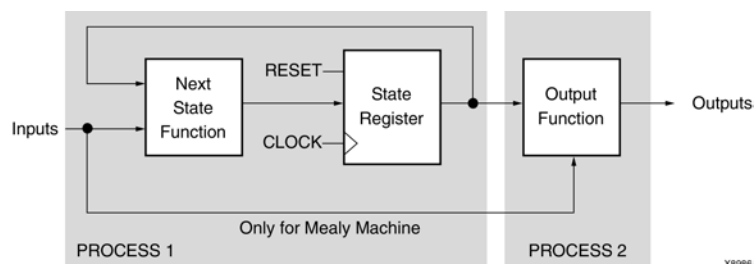

1 つの always ブロックを使用した Finite State Machine (FSM) の Verilog コード例

```
//  
// State Machine with a single always block.  
//  
module v_fsm_1 (clk, reset, x1, outp);  
    input  clk, reset, x1;  
    output outp;  
    reg    outp;  
    reg    [1:0] 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)  
            begin  
                state <= s1; outp <= 1'b1;  
            end  
        else  
            begin  
                case (state)  
                    s1: begin  
                        if (x1==1'b1)  
                            begin  
                                state <= s2;  
                                outp <= 1'b1;  
                            end  
                        else  
                            begin  
                                state <= s3;  
                                outp <= 1'b0;  
                            end  
                        end  
                    s2: begin  
                        state <= s4; outp <= 1'b1;  
                    end  
                    s3: begin  
                        state <= s4; outp <= 1'b0;  
                    end  
                    s4: begin  
                        state <= s1; outp <= 1'b0;  
                    end  
                endcase  
            end  
        end  
    end  
  
endmodule
```

2 つのプロセスブロックを使用した Finite State Machine (FSM)

出力からレジスタを削除するには、クロック同期セクションから代入文 **outp <=...** をすべて削除します。これには、に示すように 2 つのプロセスを使用します。

2 つのプロセスブロックを使用した Finite State Machine (FSM) の図



2 つのプロセスブロックを使用した Finite State Machine (FSM) のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
RESET	非同期リセット (アクティブ High)
X1	FSM の入力
OUTP	FSM の出力

2 つのプロセス ブロックを使用した Finite State Machine (FSM) の VHDL コード例

```
--
-- State Machine with two processes.
--

library IEEE;
use IEEE.std_logic_1164.all;
entity fsm_2 is
    port ( clk, reset, x1 : IN std_logic;
          outp             : OUT std_logic);
end entity;

architecture beh1 of fsm_2 is
    type state_type is (s1,s2,s3,s4);
    signal state: state_type ;
begin

    process1: process (clk,reset)
    begin
        if (reset = '1') then state <= s1;
        elsif (clk='1' and clk'Event) then
            case state is
                when s1 => if x1='1' then
                            state <= s2;
                        else
                            state <= s3;
                        end if;
                when s2 => state <= s4;
                when s3 => state <= s4;
                when s4 => state <= s1;
            end case;
        end if;
    end process process1;

    process2 : process (state)
    begin
        case state is
            when s1 => outp <= '1';
            when s2 => outp <= '1';
            when s3 => outp <= '0';
            when s4 => outp <= '0';
        end case;
    end process process2;

end beh1;
```

2 つの always ブロックを使用した Finite State Machine (FSM) の Verilog コード例

```
//
// State Machine with two always blocks.
//

module v_fsm_2 (clk, reset, x1, outp);
    input  clk, reset, x1;
    output outp;
    reg    outp;
    reg    [1:0] state;

    parameter s1 = 2'b00; parameter s2 = 2'b01;
    parameter s3 = 2'b10; parameter s4 = 2'b11;

    initial begin
        state = 2'b00;
    end

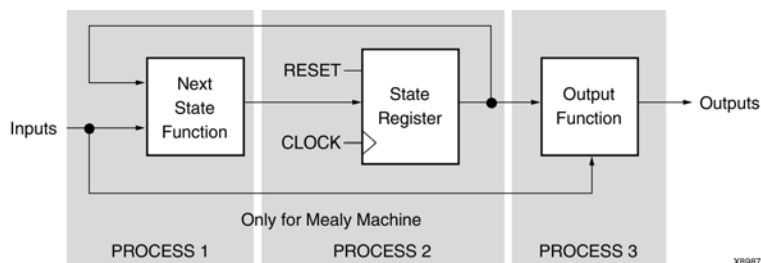
    always @(posedge clk or posedge reset)
    begin
        if (reset)
            state <= s1;
        else
            begin
                case (state)
                    s1: if (x1==1'b1)
                        state <= s2;
                        else
                            state <= s3;
                    s2: state <= s4;
                    s3: state <= s4;
                    s4: state <= s1;
                endcase
            end
        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
```

ステートレジスタから次ステート関数を分離することもできます。

3 つのプロセスブロックを使用した Finite State Machine (FSM) の図



3 つのプロセス ブロックを使用した Finite State Machine (FSM) のピンの説明

I/O ピン	説明
CLK	クロック (立ち上がりエッジ)
RESET	非同期リセット (アクティブ High)
X1	FSM の入力
OUTP	FSM の出力

3 つのプロセス ブロックを使用した Finite State Machine (FSM) の VHDL コード例

```
--
-- State Machine with three processes.
--

library IEEE;
use IEEE.std_logic_1164.all;
entity fsm_3 is
    port ( clk, reset, x1 : IN std_logic;
          outp           : OUT std_logic);
end entity;

architecture beh1 of fsm_3 is
    type state_type is (s1,s2,s3,s4);
    signal state, next_state: state_type ;
begin

    process1: process (clk,reset)
    begin
        if (reset = '1') then
            state <= s1;
        elsif (clk='1' and clk'Event) then
            state <= next_state;
        end if;
    end process process1;

    process2 : process (state, x1)
    begin
        case state is
            when s1 => if x1='1' then
                           next_state <= s2;
                       else
                           next_state <= s3;
                       end if;
            when s2 => next_state <= s4;
            when s3 => next_state <= s4;
            when s4 => next_state <= s1;
        end case;
    end process process2;

    process3 : process (state)
    begin
        case state is
            when s1 => outp <= '1';
            when s2 => outp <= '1';
            when s3 => outp <= '0';
            when s4 => outp <= '0';
        end case;
    end process process3;

end beh1;
```

3 つの always ブロックを使用した Finite State Machine (FSM) の Verilog コード例

```
//  
// State Machine with three always blocks.  
//  
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 ボックスの HDL コーディング手法

デザインには、次で生成された Electronic Data Interchange Format (EDIF) または NG ファイルが含まれることがあります。

- ・ 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デザインに関連付けるには、デザインのコードにインスタシエートする必要があります。これを XST で実行させるには、VHDL または Verilog コードにブラック ボックスのインスタシエーションを使用します。インスタシエートされたネットリストは XST では処理されず、最終の最上位ネットリストに含まれます。また、ブラック ボックスのインスタシエーションに制約を指定することも可能です。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 Register Transfer Level (RTL) モデルおよび EDIF ネットリストがある場合があります。RTL モデルはシミュレーションにしか使用できませんが、[ブラック ボックス \(BOX_TYPE\)](#) 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、「[一般制約](#)」および「[制約ガイド](#)」を参照してください。

モジュールをブラック ボックスにすると、そのモジュールのほかのインスタンスもすべてブラック ボックスになります。インスタンスに制約を指定すると、元のモジュールに指定されていた制約は無視されます。

ブラック ボックスのログ ファイル

XST ではマクロ推論の前にブラック ボックスが認識されるので、ブラック ボックスのログ ファイルはほかのマクロのものとは異なります。

```
...
Analyzing Entity <black_b> (Architecture <archi>).

WARNING:Xst:766 - black_box_1.vhd (Line 15). Generating a Black Box for component <my_block>.
Entity <black_b> analyzed. Unit <black_b> generated
....
```

ブラック ボックス関連の制約

[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化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、「[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。アップデートは、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

ブラック ボックスの VHDL コード例

```
--
-- Black Box
--

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  
//  
  
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
```

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

FPGA の最適化

この章は、次の内容が含まれます。

- ・ FPGA デバイスの最適化するための制約の使用方法
- ・ マクロ生成の説明
- ・ FPGA プリミティブのサポート

この章は、次のセクションから構成されています。

- ・ [FPGA 専用の合成オプション](#)
- ・ [マクロ生成](#)
- ・ [DSP48 ブロック リソース](#)
- ・ [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リタイミング](#)
- ・ [パーティション](#)
- ・ [エリア制約を設定した場合のスピード最適化](#)
- ・ [FPGA 最適化レポート](#)
- ・ [インプリメンテーション制約](#)
- ・ [FPGA プリミティブのサポート](#)
- ・ [コアの処理](#)
- ・ [INIT および RLOC の指定](#)
- ・ [XST での PCI™ フローの使用](#)

XST は、FPGA の合成および最適化で次の処理を実行します。

- ・ エンティティ/モジュールごとにマッピングおよび最適化
- ・ デザイン全体のグローバル最適化

このプロセスで、NGC ファイルが出力されます。

このセクションでは、次について説明します。

- ・ 合成および最適化のプロセスを制御する制約
- ・ マクロ生成
- ・ ログ ファイルに記述される情報
- ・ 合成および最適化のプロセスで使用するタイミング モデル
- ・ タイミングドリブン合成に使用可能な制約
- ・ 生成される NGC ファイルに記述される情報
- ・ プリミティブのサポート情報

FPGA 専用の合成オプション

XST は、合成プロセスを制御するオプションをサポートしています。各オプションの詳細は、「[FPGA 制約 \(タイミング制約以外\)](#)」を参照してください。

合成で使用する FPGA 専用のオプションは次のとおりです。

- ・ [BUFGCE の抽出 \(BUFGCE\)](#)
- ・ [コアの検索ディレクトリ \(-sd\)](#)
- ・ [デコーダの抽出 \(DECODER_EXTRACT\)](#)
- ・ [FSM スタイル \(FSM_STYLE\)](#)
- ・ [グローバルな最適化目標 \(-glob_opt\)](#)
- ・ [階層の維持 \(KEEP_HIERARCHY\)](#)
- ・ [論理シフタの抽出 \(SHIFT_EXTRACT\)](#)
- ・ [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 \(MULT_STYLE\)](#)
- ・ [MUX スタイル \(MUX_STYLE\)](#)
- ・ [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 \(IOB\)](#)
- ・ [プライオリティ エンコーダの抽出 \(PRIORITY_EXTRACT\)](#)
- ・ [RAM スタイル \(RAM_STYLE\)](#)
- ・ [レジスタの自動調整 \(REGISTER_BALANCING\)](#)
- ・ [レジスタの複製 \(REGISTER_DUPLICATION\)](#)
- ・ [信号のエンコード方法 \(SIGNAL_ENCODING\)](#)
- ・ [スライス パッキング \(-slice_packing\)](#)
- ・ [キャリーチェーンの使用 \(USE_CARRY_CHAIN\)](#)
- ・ [タイミング制約の書き込み \(-write_timing_constraints\)](#)
- ・ [XOR コラプス \(XOR_COLLAPSE\)](#)

マクロ生成

FPGA デバイスのマクロ ジェネレータ モジュールを使用すると、さまざまなファンクションが XST の HDL フローで使用できるようになります。これらのファンクションは推論エンジンにより HDL 記述から識別され、最適なインプリメンテーションが実行されるように、その特性がマクロ ジェネレータに渡されます。

推論されるファンクションには、加算器、アキュムレータ、カウンタ、マルチプレクサなどの単純な数値演算ブロックから、乗算器、シフトレジスタ、メモリなどの複雑なブロックまで、さまざまな種類があります。

推論されたファンクションは、Virtex® または Spartan® アーキテクチャで最高のパフォーマンスを実現できるよう最適化され、デザインに組み込まれます。また、デザインによっては、ファンクションの周囲にあるロジックと共に最適化される場合もあります。

ここでは、マクロをファンクション別に分類し、その構築および最適化の段階で使用されるリソースについて簡単に説明します。

マクロの生成は、属性を設定することにより制御できます。これらの属性は、各サブセクションにリストされています。属性の詳細は、「[デザイン制約](#)」を参照してください。

XST では、専用キャリー チェーン ロジックを使用して、多くのマクロがインプリメントされます。場合によって、キャリー チェーン ロジックにより最適な結果が得られない場合もあります。この場合、[キャリー チェーンの使用 \(USE_CARRY_CHAIN\)](#) 制約を使用すると、キャリー チェーン ロジックの使用を制御できます。

数値演算ファンクション

数値演算ファンクションには、次のエレメントがあります。

- ・ 加算器、減算器、加減算器
- ・ カスケード接続可能なバイナリ カウンタ
- ・ アキュムレータ
- ・ インクリメンタ、ディクリメンタ、インクリメンタ/ディクリメンタ
- ・ 符号付き/符号なし乗算器

XST では、キャリーの高速処理が可能なキャリー ロジック (MUXCY) が使用されるため、高速な数値演算ファンクションを実現できます。2 つの XOR ゲートで構成される積和ロジックは LUT および専用キャリー XOR (XORCY) を使用してインプリメントされます。また、専用キャリー AND (MULTAND) リソースも提供されており、高速な乗算器をインプリメントできます。

マクロ生成におけるロード可能ファンクション

ロード可能なファンクションには、次のエレメントがあります。

- ・ ロード可能アップ カウンタ、ダウン カウンタ、アップ/ダウン カウンタ
- ・ ロード可能アップ アキュムレータ、ダウン アキュムレータ、アップ/ダウン アキュムレータ

XST では、同期ロード可能、カスケード接続可能なカウンタおよびアキュムレータも推論されます。マクロの各段のカスケード接続には、高速キャリー ロジックが使用されます。同期ロードおよびカウント ファンクションは、1 つの LUT プリミティブにインプリメントされます。

アップ/ダウン カウンタおよびアキュムレータでは、パフォーマンスを向上するため、専用キャリー AND が使用されます。

マルチプレクサ

マルチプレクサには、次の 2 つのアーキテクチャがあります。

- ・ MUXF_x ベースのマルチプレクサ
- ・ 専用キャリー MUX ベースのマルチプレクサ

Virtex®-4 デバイスでは、16:1 マルチプレクサを MUXF7 プリミティブを使用して 1 つの CLB に、32:1 マルチプレクサを MUXF8 を使用して 2 つの CLB にインプリメントできます。

マルチプレクサの推論を制御するため、XST では MUXF5/MUXF6 または専用キャリー MUX のどちらを使用するかを指定できます。MUX_STYLE 属性を MUXF に設定すると MUXF_x ベースのマルチプレクサが使用され、MUXCY に設定すると専用キャリー MUX ベースのマルチプレクサが使用されます。

この属性は、マルチプレクサを定義する信号またはマルチプレクサのインスタンス名に設定します。この属性をグローバルに設定することも可能です。

MUX_EXTRACT 属性を no または force に設定すると、マルチプレクサの推論を無効にまたは強制できます。

MUX_EXTRACT 制約を使ってマルチプレクサの推論を無効にしている場合でも、MUXF_x エLEMENTが残ることがあります。こういったELEMENTは、ブール代数式の一般的なマップからのものです。

プライオリティ エンコーダ

「[プライオリティ エンコーダの HDL コーディング手法](#)」で説明されている if/elseif 構造は、1-of-n プライオリティ エンコーダでインプリメントされます。

XST では MUXCY プリミティブを使用してプライオリティ エンコーダの条件をチェーン接続し、高速のインプリメンテーションを実現します。

[プライオリティ エンコーダの抽出 \(PRIORITY_EXTRACT\)](#) 制約を使用すると、プライオリティ エンコーダ マクロの推論を有効または無効にできます。

通常、XST ではプライオリティ エンコーダは推論されないため、多数のプライオリティ エンコーダが生成されることはありません。プライオリティ エンコーダをイネーブルにするには、[プライオリティ エンコーダの抽出 \(PRIORITY_EXTRACT\)](#) 制約を force オプションで使います。

デコーダ

デコーダは、各入力値に対して 1 つのワンホット (またはワンコールド) 値が指定されているデマルチプレクサです。n ビットまたは 1-of-m デコーダには、m ビットのデータ出力および n ビットのセレクト入力があり、n と m の関係は $n \leq m \leq n \times 2$ です。

デコーダが推論されると、デコーダのサイズに応じて MUXF5 または MUXCY プリミティブがインプリメントされます。

[デコーダの抽出 \(DECODER_EXTRACT\)](#) 制約を使用して、デコーダの推論を有効または無効にします。

シフト レジスタ

XST では、次の 2 種類のシフトレジスタが生成されます。

- ・ 1 つの出力を持つシリアル シフトレジスタ
- ・ 複数の出力を持つパラレル シフトレジスタ

シフトレジスタの長さは 1 ~ 16 ビットで、次の式により決定されます。

$$= (8 \times A3) + (4 \times A2) + (2 \times A1) + A0 + 1$$

A3、A2、A1、および A0 がすべて 0 の場合 (0000) はシフトレジスタの長さは 1 ビット、すべての値が 1 の場合 (1111) は 16 ビットになります。

シリアルシフトレジスタ SRL16 では、フリップフロップが適切な幅にチェーン接続されます。

パラレルシフトレジスタの場合は、各出力がシフトレジスタの幅になります。シリアルシフトレジスタでは、その幅に対応する 1 つの出力と、次のシフトレジスタへの入力駆動されます。

シフトレジスタの抽出 (SHREG_EXTRACT) 制約を使用すると、シフトレジスタの推論を有効または無効にできます。

RAM

推論中および生成中には、次の RAM が使用可能です。

- 分散 RAM
RAM が非同期読み出しの場合は、分散 RAM が推論され、生成されます。
- ブロック RAM
RAM が同期読み出しの場合は、ブロック RAM が推論されます。この場合、ブロック RAM または分散 RAM のどちらでもインプリメントできます。
デフォルトは block RAM です。

XST で使用されるプリミティブ

このセクションは、次のデバイスに適用されます。

- Virtex®-4
- Spartan®-3

これらのデバイスの場合、XST では次の表のプリミティブが使用されます。

XST で使用されるプリミティブ

RAM	クロック エッジ	プリミティブ
シングルポート同期分散 RAM	クロックの立ち上がりエッジで動作するシングルポート分散 RAM	RAM16X1S、RAM16X2S、RAM16X4S、RAM16X8S、RAM32X1S、RAM32X2S、RAM32X4S、RAM32X8S、RAM64X1S、RAM64X2S、RAM128X1S
シングルポート同期分散 RAM	クロックの立ち下がりエッジで動作するシングルポート分散 RAM	RAM16X1S_1、RAM32X1S_1、RAM64X1S_1、RAM128X1S_1
デュアルポート同期分散 RAM	クロックの立ち上がりエッジで動作するデュアルポート分散 RAM	RAM16X1D、RAM32X1D、RAM64X1D
デュアルポート同期分散 RAM	クロックの立ち下がりエッジで動作するデュアルポート分散 RAM	RAM16X1D_1、RAM32X1D_1、RAM64X1D_1
シングルポート同期ブロック RAM	なし	RAMB4_Sn
デュアルポート同期ブロック RAM	なし	RAMB4_Sm_Sn

推論された RAM のインプリメンテーションの制御

RAM の推論を制御するため、分散 RAM またはブロック RAM (可能な場合) のどちらを使用するかを指定できます。

RAM Style (RAM_STYLE) 属性の値は次のいずれかに指定します。

- ・ ブロック RAM の場合は、block
- ・ 分散 RAM の場合は distributed

RAM スタイル (RAM_STYLE) は、次に適用します。

- ・ RAM を定義する信号
- ・ RAM のインスタンス名

RAM スタイル (RAM_STYLE) 属性は、グローバルに設定することもできます。

RAM リソースが足りない場合は、レジスタを使用して RAM を生成できます。その場合は、RAM の抽出 (RAM_EXTRACT) 制約を no に設定します。

ROM

case または if - else 文ですべての値を定数にすると ROM が推論されます。16 ワード以上の ROM (ビット幅は制限なし) のみが推論されます。たとえば、次の Hardware Description Language (HDL) の式では 16 ワード、4 ビット幅の ROM がインプリメントされます。

```
data = if address = 0000 then 0010
      if address = 0001 then 1100
      if address = 0010 then 1011
      ...
      if address = 1111 then 0001
```

また次の例のように、すべて定数で構成されている配列からも ROM が推論されます

```
type ROM_TYPE is array(15 downto 0) of std_logic_vector(3 downto 0);
constant ROM : rom_type := ("0010", "1100", "1011", ..., "0001");
...
data <= ROM(conv_integer(address));
```

ROM の抽出 (ROM_EXTRACT) 制約を使用すると、ROM の推論を無効にできます。ROM の推論を有効にするには ROM_EXTRACT の値を yes、無効にするには no に設定します。

デフォルトでは、yes に設定されています。

推論および生成中には、次の 2 種類の ROM が使用できます。

- ・ 分散 ROM
分散 ROM を使用すると、LUT、MUXF5、MUXF6、MUXF7、MUXF8 プリミティブのツリー構造を使用して、大型の ROM を小さな領域にインプリメントできます。
- ・ ブロック ROM
ブロック ROM は、ブロック RAM リソースを使用して生成されます。同期 ROM が認識されると、レジスタ付き分散 ROM として推論するか、またはブロック RAM リソースを使用して推論できます。

ROM スタイル (ROM_STYLE) 制約を使用すると、XST で推論される同期 ROM のタイプを次のように指定できます。

- ・ block : ROM が 1 つの RAM に収まる場合はブロック RAM リソースを使用してインプリメントされます。
- ・ distributed : 分散 ROM とレジスタが推論されます。
- ・ auto : ROM を使用および推論するのに最も効率的な方法が XST により選択されます。デフォルトでは auto に設定されています。

RAM スタイル (RAM_STYLE) は、VHDL 属性または Verilog メタコメントとして、個々の信号または ROM のエンティティ / モジュールに適用できます。RAM スタイル (RAM_STYLE) は、ISE® Design Suite で表示される [Process Properties] ダイアログ ボックス、またはコマンド ラインでグローバルに設定することもできます。

DSP48 ブロック リソース

XST では、次のマクロを 1 つの DSP48 ブロックに自動的にインプリメントできます。

- ・ 加算器/減算器
- ・ アキュムレータ
- ・ 乗算器
- ・ 乗算/加減算器
- ・ MAC (積和演算)

XST では、上記のマクロにレジスタが付いているものもサポートされています。

DSP48 ブロックへのマクロ インプリメンテーションは、[DSP48 の使用 \(USE_DSP48\)](#) 制約またはコマンドライン オプションでデフォルト auto に設定すると制御されます。

auto に設定すると、アキュムレータ、乗算器、乗算/加減算器、および MAC はできる限り DSP48 リソースを使用してインプリメントされますが、加減算器は DSP48 リソースにはインプリメントされません。加減算器を DSP48 にインプリメントするには、[DSP48 の使用 \(USE_DSP48\)](#) 制約またはコマンドラインの `-use_dsp48` オプションを yes に設定します。

auto モードでは、すべてのマクロが自動的に制御されます。また、使用可能な DSP48 リソースの数を[DSP 使用率 \(DSP_UTILIZATION_RATIO\)](#) 制約で制御できます。デフォルトでは、使用可能な DSP48 リソースが可能な限り使用されます。

ユーザーの指定した DSP スライス数がターゲット FPGA デバイスの 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 リソースのみが使用されて合成されます。DSP リソースが自動的に管理されないように、`DSP_UTILIZATION_RATIO` をオフにして、推論される DSP の数を確認してみてください。自動的に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。

XST では、DSP48 に最大数のレジスタを含めるなど、デフォルトで最大限のマクロ 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ています。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、各入力に 2 段のレジスタが付いている乗算器では、[キープ \(KEEP\)](#) 制約をこれらのレジスタの出力に設定して、最初のレジスタが DSP48 に含まれないようにする必要があります。

DSP48 ブロックでは、非同期セット/リセット信号の付いたレジスタがサポートされません。このため、こういったレジスタは DSP48 には含まれず、最適なデザイン パフォーマンスにはならないこともあります。[非同期から同期への変換 \(ASYNC_TO_SYNC\)](#) 制約を使用すると、デザイン全体の非同期セット/リセット信号を同期信号に置き換えることができます。これにより、DSP48 にレジスタを組み込んで結果を改善することができます。

非同期セット/リセット信号を同期信号に置換すると、生成した NGC ネットリストが最初の RTL 記述と同じではなくなります。合成したデザインが最初の仕様を満たしているかどうか必ず確認してください。詳細は、「[非同期から同期への変換 \(ASYNC_TO_SYNC\)](#)」を参照してください。

各マクロの処理に関する詳細は、「[HDL コード手法](#)」を参照してください。

内部接続されているマクロが複数含まれる場合に、それぞれが DSP48 にインプリメント可能な場合、高速 BCIN/BCOUT および PCIN/PCOUT 接続を使用して DSP48 ブロック間が内部接続されます。通常フィルタや複雑な乗算器がこの例です。

[階層の維持 \(KEEP_HIERARCHY\)](#) コマンドライン オプションが no (デフォルト) に設定されている場合、XST で階層を超えた複雑な DSP マクロおよび DSP48 チェーンを作成できます。これは ISE® Design Suite のデフォルト設定です。

ブロック RAM へのロジックのマッピング

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ブロック RAM に配置できます。

1. RTL 記述を別の階層ブロックのブロック RAM 内に入れます。
2. HDL コードまたは XCF ファイルのいずれかで、この階層ブロックに **BRAM へのロジックのマッピング (BRAM_MAP)** 制約を設定します。

XST では、ブロック RAM に配置するロジックを自動的に判断できないので注意してください。

ロジックを別のブロックにする場合、次の条件を満たす必要があります。

- ・ すべての出力がレジスタを介するようになる。
- ・ 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。
- ・ ブロックに複数のソースまたはトライステート バスが含まれない。
- ・ 中間信号に **キープ (KEEP)** 制約を使用していない。

ブロック RAM へのロジックのマッピングは、アドバンス合成段階で実行されます。上記の条件が 1 つでも満たされない場合は、ロジックはブロック RAM にマッピングされず、警告メッセージとその理由が示されます。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。

ブロック RAM へのロジックのマッピングのログ ファイル

このセクションには、ブロック RAM にロジックをマッピングする際のログ ファイルが含まれます。

- ・ ブロック RAM へのロジックのマッピングのログ ファイル例 1
- ・ ブロック RAM へのロジックのマッピングのログ ファイル例 2

ブロック RAM へのロジックのマッピングのログ ファイル例 1

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

Synthesizing Unit <logic_bram_1>.
  Related source file is "bram_map_1.vhd".
  Found 4-bit register for signal <RES>.
  Found 4-bit adder for signal <$n0001> created at line 29.
  Summary:
    inferred    4 D-type flip-flop(s).
    inferred    1 Adder/Subtractor(s).
Unit <logic_bram_1> synthesized.

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

...
Entity <logic_bram_1> mapped on BRAM.
...
=====
HDL Synthesis Report

Macro Statistics
# Block RAMs                : 1
256x4-bit single-port block RAM : 1
...
=====
```


ブロック RAM へのロジックのマッピングのログ ファイル例 2

```
...
=====
*                               Advanced HDL Synthesis                               *
=====
...
INFO:Xst:1789 - Unable to map block <no_logic_bram> on BRAM.
               Output FF <RES> must have a synchronous reset.
```

ブロック RAM へのロジックのマッピングのコード例

コード例は、本書が作成された時点のものです。アップデートは、
[ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip](http://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip) からダウンロードしてください。

単一ブロック RAM プリミティブ内の定数付き 8 ビット加算器の VHDL コード例

```
--
-- The following example places 8-bit adders with
-- constant in a single block RAM primitive
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity logic_bram_1 is
port (clk, rst : in std_logic;
      A,B : in unsigned (3 downto 0);
      RES : out unsigned (3 downto 0));

    attribute bram_map: string;
    attribute bram_map of logic_bram_1: entity is "yes";

end logic_bram_1;

architecture beh of logic_bram_1 is
begin

    process (clk)
    begin
        if (clk'event and clk='1') then
            if (rst = '1') then
                RES <= "0000";
            else
                RES <= A + B + "0001";
            end if;
        end if;
    end process;

end beh;
```

単一ブロック RAM プリミティブ内の定数付き 8 ビット加算器の Verilog コード例

```
//
// The following example places 8-bit adders with
// constant in a single block RAM primitive
//

(* bram_map="yes" *)
module v_logic_bram_1 (clk, rst, A, B, RES);

    input  clk, rst;
    input  [3:0] A, B;
    output [3:0] RES;
    reg    [3:0] RES;

    always @(posedge clk)
    begin
        if (rst)
            RES <= 4'b0000;
        else
            RES <= A + B + 8'b0001;
        end
    end

endmodule
```

非同期リセットの VHDL コード例

```
--
-- In the following example, an asynchronous reset is used and
-- so, the logic is not mapped onto block RAM
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity logic_bram_2 is
port (clk, rst : in std_logic;
      A,B       : in unsigned (3 downto 0);
      RES       : out unsigned (3 downto 0));

    attribute bram_map : string;
    attribute bram_map of logic_bram_2 : entity is "yes";

end logic_bram_2;

architecture beh of logic_bram_2 is
begin

    process (clk, rst)
    begin
        if (rst='1') then
            RES <= "0000";
        elsif (clk'event and clk='1') then
            RES <= A + B + "0001";
        end if;
    end process;

end beh;
```

非同期リセットの Verilog コード例

```
//  
// In the following example, an asynchronous reset is used and  
// so, the logic is not mapped onto block RAM  
//  
  
(* bram_map="yes" *)  
module v_logic_bram_2 (clk, rst, A, B, RES);  
  
    input  clk, rst;  
    input  [3:0] A, B;  
    output [3:0] RES;  
    reg    [3:0] RES;  
  
    always @(posedge clk or posedge rst)  
    begin  
        if (rst)  
            RES <= 4'b0000;  
        else  
            RES <= A + B + 8'b0001;  
        end  
    end  
  
endmodule
```

フリップフロップのリタイミング

フリップフロップのリタイミングとは、タイミングを向上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。

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。

フリップフロップのリタイミングをすると、次が発生することがあります。

- ・ フリップフロップの数がかなり増加します。
- ・ フリップフロップの一部が削除されます。

どちらが発生しても、デザインの動作に変更はなく、タイミング遅延のみが変化します。

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。タイミング制約が満たされた場合、またはタイミングが向上しない場合は、それ以上のリタイミングは行われません。

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、次のような制限があります。

- ・ 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、フリップフロップは順方向には移動されません。
- ・ 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、フリップフロップは逆方向には移動されません。
- ・ インスタンス化されたフリップフロップは、[Optimize Instantiated Primitives] 制約またはこのコマンドライン オプションが yes になっている場合にのみ移動されます。
- ・ フリップフロップは、[Optimize Instantiated Primitives] 制約またはこのコマンドライン オプションが yes になっている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。
- ・ セットとリセット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

XST では、インクリメンタル合成の代わりにパーティションがサポートされています。インクリメンタル合成はサポートされなくなりました。このため、incremental_synthesis および resynthesize 制約もサポートされません。パーティションの詳細は、ISE 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 ではエリア制約を設定したデザインでタイミング最適化を実行できます。このオプションは、次のように指定します。

- ・ LUT と FF ペア の使用率 (Virtex®-5 デバイス)
- ・ **スライス (LUT-FF ペア) 使用率 (SLICE_UTILIZATION_RATIO)** (その他の FPGA デバイス)

ISE® Design Suite で次のように定義します。

[Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスを表示して、[XST Synthesis Options] をクリックします。

デフォルトでは、選択したデバイス サイズの 100% に設定されています。

この制約は下位レベルの合成のみに適用され、推論プロセスには影響しません。

この制約を設定すると、まずエリアが概算され、指定のエリア制約が満たされてから、制約で指定した値を超えないようにタイミング最適化が行われます。デザインが要求よりも大きい場合は、まずエリアを削減し、エリア制約が満たされてからタイミング最適化が行われます。

エリア制約を設定した場合のスピード最適化の例 1 (100%)

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、最初の概算ではデバイスの 102% を占めることが示されています。XST により最適化が実行され、95% に削減されました。

```
...
=====
*
*                               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.

=====
```

エリア制約を設定した場合のスピード最適化の例 2 (70%)

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、この制約を満たすことができなかったため、警告メッセージが表示されています。

```
...
=====
*
*                               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
...
=====
...
```

メモ： (+5) は、エリア制約の最大マージンを示します。これは、エリア制約が満たされない場合、エリア最適化において制約と実際のエリアとの差が 5% 以下であれば、そのエリアを超えない範囲でタイミング最適化が実行されることを意味します。

エリア制約を設定した場合のスピード最適化の例 3 (55%)

次の例では、エリア制約が 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% なので、エリア制約は満たされたものとしてタイミング最適化が実行されます。

```
...
=====
*
*                               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、デザインの特定期ブロックに対して設定できます。合計スライス数のパーセントまたはスライスの絶対数で指定できます。

FPGA 最適化のレポート セクション

デザインの最適化中、次の事項がレポートされます。

- ・ 等価フリップフロップの削除

データ ピンと制御ピンが同じ場合、2 つのフリップフロップは等価であると判断されます。

- ・ レジスタの複製

レジスタの複製は、タイミングを向上させるためか、MAX_FANOUT 制約を満たすために行われます。[レジスタの複製 \(REGISTER DUPLICATION\)](#) 制約を使用すると、レジスタの複製を無効にできます。

デザイン最適化のレポート例

```
Starting low level synthesis ...
Optimizing unit <down4cnt> ...
Optimizing unit <doc_readwrite> ...
...
Optimizing unit <doc> ...
Building and optimizing final netlist ...
The FF/Latch <doc_readwrite/state_D2> in Unit <doc> is equivalent to the following 2 FFs/Latches,
which will be removed : <doc_readwrite/state_P2> <doc_readwrite/state_M2>Register
doc_reset_1_reset_out has been replicated 2 time(s)
Register wr_1 has been replicated 2 time(s)
```

セル使用率レポート

「Final Report」(最終レポート)の「Cell Usage」(セル使用率)セクションには、デザインで使用されたプリミティブの合計数が示されます。プリミティブは、次のグループに分類されます。

- ・ BEL のセル使用率
- ・ フリップフロップとラッチのセル使用率
- ・ RAM のセル使用率
- ・ シフタのセル使用率
- ・ トライステートのセル使用率
- ・ クロック バッファのセル使用率
- ・ IO バッファのセル使用率
- ・ 論理セル使用率
- ・ その他のセル使用率

BEL のセル使用率

「Final Report」(最終レポート)の「Cell Usage」(セル使用率)セクションの BEL グループには、ターゲット FPGA デバイスファミリの基本エレメントである次のようなロジックセルすべてが含まれます。

- ・ LUT
- ・ MUXCY
- ・ MUXF5
- ・ MUXF6
- ・ MUXF7
- ・ MUXF8

フリップフロップとラッチのセル使用率

「Final Report」(最終レポート)の「Cell Usage」(セル使用率)セクションのフリップフロップとラッチのグループには、ターゲット デバイス ファミリの基本エレメントである次のようなロジック セルすべてが含まれます。

- ・ FDR
- ・ FDRE
- ・ LD

RAM のセル使用率

「Final Report」(最終レポート)の「Cell Usage」(セル使用率)セクションの RAM グループには、RAM すべてが含まれます。

シフタのセル使用率

「Final Report」(最終レポート)の「Cell Usage」(セル使用率)セクションのシフタ グループには、FPGA デバイス プリミティブを使用する次のようなシフトレジスタがすべて含まれます。

- ・ TSRL16
- ・ SRL16_1
- ・ SRL16E
- ・ SRL16E_1
- ・ SRLC

トライステートのセル使用率

「Final Report」(最終レポート)の「Cell Usage」(セル使用率)セクションのトライステート グループには、次のようなトライステート プリミティブがすべて含まれます。

BUFT

クロック バッファのセル使用率

「Final Report」(最終レポート)の「Cell Usage」(セル使用率)セクションのクロック バッファには、次のようなクロック バッファがすべて含まれます。

- ・ BUFG
- ・ BUFGP
- ・ BUFGDLL

IO バッファのセル使用率

「Final Report」(最終レポート)の「Cell Usage」(セル使用率)セクションの I/O バッファには、次のようなクロック バッファ以外の I/O バッファがすべて含まれます。

- ・ IBUF
- ・ OBUF
- ・ IOBUF
- ・ OBUFT
- ・ IBUF_GTL ...

論理セル使用率

「Final Report」(最終レポート)の「Cell Usage」(セル使用率)セクションの論理 (LOGICAL) グループには、基本エレメント以外のすべての論理セル プリミティブが含まれます。

- ・ AND2
- ・ OR2 ...

その他のセル使用率

「Final Report」(最終レポート)の「Cell Usage」(セル使用率)セクションのその他 (OTHER) グループには、上記のグループに属さないすべてのセルが含まれます。

セル使用率のレポート例

```
=====  
...  
Cell Usage :  
# BELS                      : 70  
#   LUT2                     : 34  
#   LUT3                     : 3  
#   LUT4                     : 34  
# FlipFlops/Latches        : 9  
#   FDC                     : 8  
#   FDP                     : 1  
# Clock Buffers            : 1  
#   BUFGP                   : 1  
# IO Buffers               : 24  
#   IBUF                   : 16  
#   OBUF                   : 8  
=====
```

XST により予測されたスライス、フリップフロップ、IOB、BRAM などの数が示されます。このセクションは、MAP で生成されるレポートと類似しています。

レポートには、デザインで使用されているクロックの数、クロック バッファの種類、およびロードの数が表示されます。

また、デザインに含まれる非同期セット/リセット信号の数、各信号のバッファ方法、ロード数なども表示されます。

タイミング レポート

XST では合成の最後に詳細なタイミング情報がレポートされます。タイミング レポートには、ネットリストの 4 つの使用可能なドメインに関する情報が表示されます。

- ・ レジスタからレジスタ
- ・ 入力からレジスタ
- ・ レジスタから出力パッド
- ・ 入力パッドから出力パッド

タイミング レポートの例

これらのタイミングの値は、あくまで合成での概算にすぎないので、正確なタイミング情報は配置配線後の TRACE レポートを参照してください。

Clock Information:

Clock Signal	Clock buffer (FF name)	Load
CLK	BUFGP	11

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

Timing Summary:

Speed Grade: -12

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 combinational path delay: 4.473ns

Timing Detail:

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 Delay	Net Delay	Logical Name (Net Name)
FDC:C->Q	8	0.272	0.642	MACHINE/current_state_FFd3 (MACHINE/current_state_FFd3)
LUT3:I0->O	3	0.147	0.541	Ker81 (clkenable)
LUT4_D:I1->O	1	0.147	0.451	sixty/msbce (sixty/msbce)
LUT3:I2->O	1	0.147	0.000	sixty/msbcount/qoutsig_3_rstpot (N43)
FDC:D		0.297		sixty/msbcount/qoutsig_3
Total		2.644ns (1.010ns logic, 1.634ns route) (38.2% logic, 61.8% route)		

タイミング レポートのタイミング サマリ セクション

このセクションには、4 つのドメインすべてのタイミング パスに関するサマリ情報が示されます。

- ・ クロックからクロックまでのパス
Minimum period: 7.523ns (Maximum Frequency: 132.926MHz)
- ・ すべてのプライマリ出力からシーケンシャル エlementまでの最大パス
Minimum input arrival time before clock: 8.945ns
- ・ シーケンシャル エlementからすべてのプライマリ出力までの最大パス
Maximum output required time before clock: 14.220ns
- ・ 入力から出力までの最大パス
Maximum combinational path delay: 10.899ns

ドメインに該当するパスがない場合は、「No path found」と記述されます。

タイミング レポートの Detail セクション

このセクションには、次の各領域で最もクリティカルなパスに関する情報が詳細に記述されます。

- ・ パスの開始点
- ・ パスの終点
- ・ パスの最大遅延
- ・ スラック

始点と終点は、次のいずれかになります。

- ・ クロック (立ち上がり/立ち下りのパルス付き)
- ・ ポート

Path from Clock 'sysclk' rising to Clock 'sysclk' rising : 7.523ns (Slack: -7.523ns)

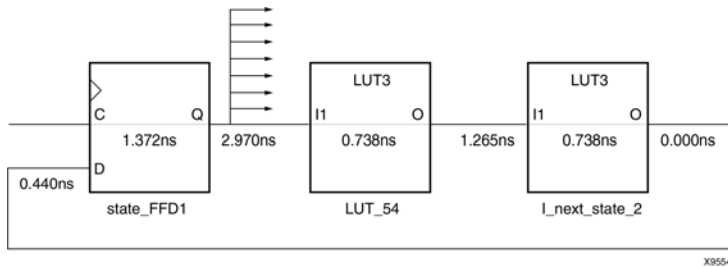
パスの詳細には、次の情報が含まれます。

- ・ セル タイプ
- ・ このゲートの入力と出力
- ・ 出力のファンアウト
- ・ ゲート遅延
- ・ ネット遅延の概算
- ・ インスタンス名

階層ブロックの始めには「begin scope」と記述され、ブロックの終わりには「end scope」と記述されます。

タイミング レポートの回路図

前述のレポートは、次の回路図に対応しています。



タイミング レポートのパスとポート

また、タイミング レポートのセクションでは、解析されたパス数およびポート数が示されます。XST の実行時にタイミング 制約が含まれる場合は、エラーが発生したパス数およびポート数も表示されます。エラーが発生したパス数は、デザインに含まれるタイミング エラー数を示し、エラーが発生したポート数は、これらがデザインでどのように配置されているかを示します。タイミング レポートのポート数は、タイミング制約のデスティネーション エLEMENT数を示します。

たとえば、次のタイミング制約を使用するとします。

```
TIMESPEC "TSidentifier"=FROM "source_group" TO "dest_group" value units;
```

この場合、ポート数はデスティネーション グループに含まれるELEMENT数に対応します。

XST でエラーが発生したパス数が 100 であると表示されているのに、エラーが発生したデスティネーション ポートはフリップフロップ 2 個のみであるというような場合があります。この場合、これらの 2 個フリップフロップに関する記述を解析するだけで十分です。

インプリメンテーション制約

XST は、Hardware Description Language (HDL) または制約ファイルの属性 (LOC など) から生成されたインプリメンテーション制約を NGC ファイルに出力します。

キープ (KEEP) プロパティは、最大ファンアウトの制御または最適化を目的として、バッファ挿入プロセスにより生成されます。

FPGA デバイス プリミティブのサポート

XST では、デバイスのプリミティブを VHDL/Verilog コードに直接インスタンスエートできます。次のようなプリミティブは、インスタンスエートすると HDL デザインに手動で挿入できます。

- ・ MUXCY_L
- ・ LUT4_L
- ・ CLKDLL
- ・ RAMB4_S1_S16
- ・ IBUFG_PCI33_5
- ・ NAND3b2

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されます。
- ・ デフォルトでは XST で最適化されません。
- ・ 最終的な NGC ファイルに記述されます。

[Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスで [Xilinx Specific Options] ページにある [Optimize Instantiated Primitives] をオンにすると、インスタンス化されたプリミティブを最適化して結果を向上させることができます。ほとんどのプリミティブにタイミング情報があり、XST で効果的なタイミングドリブン最適化が実行されます。

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。詳細については、『ライブラリ ガイド』を参照してください。

属性を使用したプリミティブの生成

属性により生成できるプリミティブもあります。

- ・ **バッファ タイプ (BUFFER_TYPE)** をプライマリ入力または内部信号に設定すると、BUFGDLL、IBUFG、BUFR、または BUFGP 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、「[セーフ FSM 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。

ブラック ボックスとザイリンクス 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。たとえば次のログ ファイル例では、MUX5 が MUX51 に変更されています。

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

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。ブラック ボックスとザイリンクス 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

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化をシンプルにするため、VHDL および Verilog の両方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。

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

VHDL の場合、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;
```

このパッケージのソース コードは、XST のインストール ディレクトリにある `vhdl¥src¥unisim¥unisim_vcomp.vhd` ファイルに含まれています。

Verilog 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 インスタンス化のガイドライン

プリミティブをインスタンス化の際は、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 エレメントは UNISIM ライブラリで次のように宣言されています。

```
component ODDR  
generic  
  (DDR_CLK_EDGE : string := "OPPOSITE_EDGE";  
   INIT : bit := '0';  
   SRTYPE : string := "SYNC");  
  
port(Q : out std_ulogic;  
     C : in std_ulogic;  
     CE : in std_ulogic;  
     D1 : in std_ulogic;  
     D2 : in std_ulogic;  
     R : in std_ulogic;  
     S : in std_ulogic);  
end component;
```

このプリミティブをインスタンス化する場合、DDR_CLK_EDGE および SRTYPE 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、インスタンス化で INIT を使用できます。INIT を最終ネットリストに渡すには、次の 2 つの方法があります。

- ・ INIT 属性をインスタンス化したプリミティブに設定します。
- ・ VHDL のジェネリックまたは Verilog のパラメータを使用して INIT を渡します。このようにすると合成とシミュレーションに同じコードを使用できるので、ザイリンクスではこの方法をお勧めしています。

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

UNISIM ライブラリの各プリミティブでは、**ボックス タイプ (BOX_TYPE)** 属性が `primitive` に設定されているため、HDL 合成中にインスタンス化されたデバイス プリミティブに関するメッセージは表示されません。

プリミティブではないブロックをデザインにインスタンス化して、ブロックにロジック記述がない場合、またはブロックにロジック記述があるが、**ボックス タイプ (BOX_TYPE)** 制約が user_black_box に設定されている場合には、ログ ファイルに次のような警告メッセージが表示されます。

```
...
Analyzing Entity <black_b> (Architecture <archi>).
WARNING : (VHDL_0103). c:\jm\des.vhd (Line 23). Generating a Black Box for component <my_block>.
Entity <black_b> analyzed. Unit <black_b> generated.
...
```

プリミティブ関連の制約

- ・ **ボックス タイプ (BOX_TYPE)**
- ・ 処理せずに HDL から NGC に渡される PAR 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。アップデートは、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

VHDL コード例 (INIT 制約で INIT 値指定)

```
--
-- Passing an INIT value via the INIT constraint.
--

library ieee;
use ieee.std_logic_1164.all;

library unisim;
use unisim.vcomponents.all;

entity primitive_1 is
    port(I0,I1 : in std_logic;
         O : out std_logic);
end primitive_1;

architecture beh of primitive_1 is

    attribute INIT: string;
    attribute INIT of inst: label is "1";

begin

    inst: LUT2 port map (I0=>I0,I1=>I1,O=>O);

end beh;
```

Verilog コード例 (INIT 制約で INIT 値指定)

```
//
// Passing an INIT value via the INIT constraint.
//

module v_primitive_1 (I0,I1,O);
    input I0,I1;
    output O;

    (* INIT="1" *)
    LUT2 inst (.I0(I0), .I1(I1), .O(O));

endmodule
```

VHDL コード例 (ジェネリック文で INIT 値指定)

```
--
-- Passing an INIT value via the generics mechanism.
--

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;
```

Verilog コード例 (パラメータ文で INIT 値指定)

```
//
// Passing an INIT value via the parameters mechanism.
//

module v_primitive_2 (I0,I1,O);
    input I0,I1;
    output O;

    LUT2 #(4'h1) inst (.I0(I0), .I1(I1), .O(O));

endmodule
```

Verilog コード例 (defparam で INIT 値指定)

```
//
// Passing an INIT value via the defparam mechanism.
//

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。UniMacro ライブラリは、Virtex®-4、Virtex-5、およびそれ以降の新規デバイスでサポートされます。詳細については、該当するデバイスの『ライブラリ ガイド』を参照してください。

VHDL の場合、ソースコードでパッケージ vcomponents を使用してライブラリ unisim を宣言します。

```
library unimacro;
use unimacro.vcomponents.all;
```

このパッケージのソースコードは、XST 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 ライブラリはあらかじめコンパイルされており、XST により自動的にデザインと関連付けられます。

コアの処理

デザインに Electronic Data Interchange Format (EDIF) または NGC ファイル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エリア使用率の制御に使用されます。この機能は、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、[Synthesis Options] ページにある [Read Cores] でオン/オフを切り替えます。コマンドラインの場合、run コマンドで `-read_cores` オプションを使用します。optimize オプションも指定できます。このオプションを使用すると、コアの処理が可能になり、コアのネットリストをデザインに統合できます。XST では、デフォルトでコアが読み込まれます。

コアの処理の VHDL コード例

次の VHDL の例では、my_add ブロックはブラック ボックスとして記述されている加算器で、このデザインのネットリストは CORE Generator™ で生成されています。

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity read_cores is
  port(
    A, B : in std_logic_vector (7 downto 0);
    a1, b1 : in std_logic;
    SUM : out std_logic_vector (7 downto 0);
    res : out std_logic);
end read_cores;

architecture beh of read_cores is
  component my_add
  port (
    A, B : in std_logic_vector (7 downto 0);
    S : out std_logic_vector (7 downto 0));
  end component;

begin
  res <= a1 and b1;
  inst: my_add port map (A => A, B => B, S => SUM);
end beh;
```

[Read Cores] のオン/オフ

[Read Cores] がオフの場合、組み合わせパスの最大遅延は 6.639ns (クリティカル パスは単純な AND ファンクションを通過)、使用されるエリアは 1 スライスと予測されます。

[Read Cores] がオンの場合、下位レベルの合成時に次のメッセージが表示されます。

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

Launcher: Executing edif2ngd -noa "my_add.edn" "my_add.ngo"
INFO:NgdBuild - Release 6.1i - edif2ngd G.21
INFO:NgdBuild - Copyright (c) 1995-2003 Xilinx, Inc. All rights reserved.
Writing the design to "my_add.ngo"...
Loading core <my_add> for timing and area information for instance <inst>.

=====
...
```

この場合、組み合わせパスの最大遅延は 8.281ns、使用されるエリアは 5 スライスと予測されます。

デフォルトでは、コアの Electronic Data Interchange Format (EDIF) または NGC ファイルは、作業中のプロジェクトのディレクトリから読み込まれます。コア ファイルがプロジェクト ディレクトリにない場合は、[コアの検索ディレクトリ \(-sd\)](#) オプションでコアのディレクトリを指定する必要があります。

INIT および RLOC の指定

UNISIM ライブラリを使用すると、LUT コンポーネントを直接 Hardware Description Language (HDL) コードにインスタンス化できます。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、インスタンスに [RLOC](#) 制約を設定します。

INIT でファンクションを定義するのが不都合な場合は、ファンクションを個別のブロックとして VHDL または Verilog で記述し、LUT にマップする方法もあります。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。XST では、Synplicity でサポートされる XC_MAP 制約が自動的に認識されます。

LUT_MAP 制約を使用して INIT 値を渡すコード例

コード例は、本書が作成された時点のものです。アップデートは、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

次に、LUT_MAP 制約を使用して INIT 値を渡す例を示します。

- ・ LUT_MAP 制約を使用して INIT 値を渡す VHDL コード例
- ・ LUT_MAP 制約を使用して INIT 値を渡す Verilog コード例

これらの例では、top ブロックに and_one および and_two で記述される 2 つの AND ゲートがインスタンス化されています。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。詳細は、「[単一 LUT へのエンティティのマップ \(LUT_MAP\)](#)」を参照してください。

LUT_MAP 制約を使用して INIT値を渡すVHDL コード例

```
--
-- Mapping on LUTs via LUT_MAP constraint
--

library ieee;
use ieee.std_logic_1164.all;
entity and_one is
    port (A, B : in std_logic;
          REZ : out std_logic);

    attribute LUT_MAP: string;
    attribute LUT_MAP of and_one: entity is "yes";
end and_one;

architecture beh of and_one is
begin
    REZ <= A and B;
end beh;

-----

library ieee;
use ieee.std_logic_1164.all;
entity and_two is
    port(A, B : in std_logic;
          REZ : out std_logic);

    attribute LUT_MAP: string;
    attribute LUT_MAP of and_two: entity is "yes";
end and_two;

architecture beh of and_two is
begin
    REZ <= A or B;
end beh;

-----

library ieee;
use ieee.std_logic_1164.all;
entity inits_rlocs_1 is
    port(A,B,C : in std_logic;
          REZ : out std_logic);
end inits_rlocs_1;

architecture beh of inits_rlocs_1 is

    component and_one
    port(A, B : in std_logic;
          REZ : out std_logic);
    end component;

    component and_two
    port(A, B : in std_logic;
          REZ : out std_logic);
    end component;

    signal tmp: std_logic;
begin
    inst_and_one: and_one port map (A => A, B => B, REZ => tmp);
    inst_and_two: and_two port map (A => tmp, B => C, REZ => REZ);
end beh;
```

LUT_MAP 制約を使用して INIT 値を渡す Verilog コード例

```
//  
// Mapping on LUTs via LUT_MAP constraint  
//  
(* 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_inits_rlocs_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
```

フリップフロップの INIT 値を指定するコード例

コード例は、本書が作成された時点のものです。アップデートは、
http://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

ファンクションが 1 つの LUT にマップできない場合、エラー メッセージが表示され、合成が停止します。RTL レベルでフリップフロップまたはシフトレジスタの INIT 値を定義する場合は、信号宣言の段階で初期値を指定します。この値が合成中に無視されることなく、フリップフロップまたはシフトレジスタに INIT 制約が設定されて、最終ネットリストに含まれます。

次の例では、信号 tmp に対して 4 ビットレジスタが推論されます。

推論されたレジスタには INIT 値 1011 が設定され、最終ネットリストに含まれます。

フリップフロップの INIT 値を指定した VHDL コード例

```
--
-- Specification on an INIT value for a flip-flop, described at RTL level
--

library ieee;
use ieee.std_logic_1164.all;

entity inits_rlocs_2 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_2;

architecture beh of inits_rlocs_2 is signal
    tmp: std_logic_vector(3 downto 0):="1011";
begin

    process (CLK)
    begin
        if (clk'event and clk='1') then
            tmp <= DI;
        end if;
    end process;

    DO <= tmp;

end beh;
```

フリップフロップの INIT 値を指定する Verilog コード例

```
//
// Specification on an INIT value for a flip-flop,
// described at RTL level
//

module v_inits_rlocs_2 (clk, di, do);
    input  clk;
    input  [3:0] di;
    output [3:0] do;
    reg    [3:0] tmp;

    initial begin
        tmp = 4'b1011;
    end

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

    assign do = tmp;

endmodule
```

フリップフロップの INIT 値および RLOC 値を指定するコード例

コード例は、本書が作成された時点のものです。アップデートは、
[ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip](http://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip) からダウンロードしてください。

レジスタが推論され、チップの特定の位置に配置されるようにするには、次のコード例のように tmp 信号に **RLOC** 制約を設定します。

XST では、この制約を最終ネットリストに含めます。この制約は、レジスタでサポートされるほか、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場合は、推論された RAM でもサポートされます。

フリップフロップの INIT 値および RLOC 値を指定する VHDL コード例

```
--
-- Specification on an INIT and RLOC values for a flip-flop, described at RTL level
--

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;
```

フリップフロップの INIT 値を指定した Verilog コード例

```
//
// Specification on an INIT and RLOC values for a flip-flop,
// described at RTL level
//

module v_inits_rlocs_3 (clk, di, do);
    input  clk;
    input  [3:0] di;
    output [3:0] do;
    (* RLOC="X3Y0 X2Y0 X1Y0 X0Y0" *)
    reg    [3:0] tmp;

    initial begin
        tmp = 4'b1011;
    end

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

    assign do = tmp;
endmodule
```

XST での PCI フローの使用

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。指定する場合は、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、[Synthesis Options] ページの [Case] プロパティを変更してください。
- ・ 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。
デフォルトでは、[Maintain] になっています。指定する場合は、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、[Synthesis Options] ページの [Case] プロパティを変更してください。
- ・ 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 [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 [Keep Hierarchy] で設定できます。
- ・ 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 [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 [Equivalent Register Removal] から設定できます。

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高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くするには、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。
- ・ [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] 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と、デザインのユーザーが作成した部分に対してロジック最適化が実行されてタイミング制約が満たされなくなったり、マップ中にエラーが発生することがあります。[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。

CPLD の最適化

この章では、CPLD の合成オプションおよびマクロ生成のインプリメンテーションについて説明します。この章は、次のセクションから構成されています。

- ・ CPLD 合成オプション
- ・ マクロ生成のインプリメンテーション
- ・ CPLD 合成のログ ファイルの解析
- ・ CPLD 合成の制約
- ・ CPLD 合成結果の向上

CPLD 合成オプション

XST では、CPLD フィッタ用に NGC ファイルが生成されます。

CPLD を合成するための一般的な XST フローは、次のとおりです。

1. [VHDL] または [Verilog] を選択します。
2. マクロの推論
3. モジュールの最適化
4. NGC ファイルの生成

このセクションでは、サポートされている CPLD ファミリーと、CPLD 合成にのみ関連した XST オプションについて説明します。これらのオプションは、ISE® Design Suite の [Process Properties] ダイアログ ボックスから設定できます。

CPLD 合成でサポートされるデバイス

XST でサポートされている CPLD デバイスは次の 5 つです。

- ・ CoolRunner™ XPLA3
- ・ CoolRunner-II
- ・ XC9500
- ・ XC9500XL

CoolRunner XPLA3 および XC9500XL デバイス ファミリーの合成では、クロック イネーブルが処理されます。クロック イネーブルは、許可するか無効にできます。無効にした場合は、同等のロジックに置換されます。また、カウンタなどのクロック イネーブルを使用するマクロが選択可能かどうかは、デバイス ファミリーによって異なります。クロック イネーブルを使用するカウンタは、CoolRunner XPLA3、XC9500XL ファミリーでは選択できますが、XC9500 ファミリーでは使用不可で、同等のロジックに置き換えられます。

CPLD 合成オプションの設定

CPLD 合成オプションは、ISE® Design Suite の [Process Properties] の [Synthesis Options] ページから設定できます。詳細は、「[CPLD 制約 \(タイミング以外\)](#)」を参照してください。

- ・ [階層の維持 \(KEEP_HIERARCHY\)](#)
- ・ [マクロの保持 \(-pld_mp\)](#)
- ・ [XOR の保持 \(-pld_xp\)](#)
- ・ [等価レジスタの削除 \(EQUIVALENT_REGISTER_REMOVAL\)](#)
- ・ [クロック イネーブル \(-pld_ce\)](#)
- ・ [WYSIWYG \(-wysiwyg\)](#)
- ・ [削減なし \(NOREDUCE\)](#)

マクロ生成のインプリメンテーション

XST では次のマクロが処理されます。

- ・ 加算器
- ・ 減算器
- ・ 加減算器
- ・ 乗算器
- ・ コンパレータ
- ・ マルチプレクサ
- ・ カウンタ
- ・ 論理シフタ
- ・ レジスタ (フリップフロップおよびラッチ)
- ・ XOR

マクロの生成は、[Macro Preserve] オプションで設定します。このオプションには、次の 2 つの値があります。

- ・ **yes**
マクロ生成が許可されます。
- ・ **no**
マクロ生成が禁止されます。

一般的なマクロ生成のフローは次のとおりです。

1. Hardware Description Language (HDL) ではマクロが推論され、下位合成に渡されます。
2. 下位合成で、マクロのインプリメンテーションに必要なリソースによって、マクロとして処理されるかどうかが決まります。

マクロとして処理されるものは、内部マクロ生成機能で生成されます。マクロとして処理されないものは、HDL 合成で等価ロジックに置き換えられるか、またはコンポーネントブロックに分解され、コンポーネントが最初のマクロと比較して使用するリソースが少ないマクロとなるように処理されます。後者の場合、この新しいより小型の方のマクロが XST でマクロとして処理されることもあります。たとえば、クロック イネーブル (CE) を持つフリップフロップ マクロは、XC9500 にマップするとマクロとして処理されませんが、HDL 合成により次の 2 つのマクロが生成されます。

- ・ クロック イネーブル信号のないフリップフロップ マクロ
- ・ クロック イネーブル機能をインプリメントする MUX マクロ

生成されたマクロは個別に最適化され、周辺のロジックと結合されます。このように最適化すると、大型のコンポーネントで良い結果が得られます。

CPLD 合成のログ ファイルの解析

XST の CPLD 合成に関する出力ログは、次のメッセージの後にあります。

Low Level Synthesis

XST で出力されるログ ファイルには、次の情報が含まれています。

- ・ ユニットの最適化の進捗状況を表示

Optimizing unit *unit_name* ...

- ・ ユニット最適化に関する情報、警告、エラー メッセージ
 - 代理式のシェーピング機能が使用されている場合 (XC9500 デバイスのみ)

Collapsing ...

- 等価フリップフロップが削除された場合

Register *#1* equivalent to *#2* has been removed

- XST でユーザー指定の制約が処理された場合

implementation constraint: *constraint_name*[=*value*]: *signal_name*

- ・ 最終統計

Final Results

Top Level Output file name : *file_name*

Output format : ngc

Optimization goal : {area | speed}

Target Technology : {9500 | 9500x1 | 9500xv | xpla3 | xbr | cr2s}

Keep Hierarchy : {yes | soft | no}

Macro Preserve : {yes | no}

XOR Preserve : {yes | no}

Design Statistics

NGC Instances: *nb_of_instances*

I/Os: *nb_of_io_ports*

Macro Statistics

FSMs: *nb_of_FSMs*

Registers: *nb_of_registers*

Tristates: *nb_of_tristates*

Comparators: *nb_of_comparators*

n-bit comparator {equal | not equal | greater | less | greatequal | lessequal}:

nb_of_n_bit_comparators

Multiplexers: *nb_of_multiplexers*

n-bit m-to-1 multiplexer :

nb_of_n_bit_m_to_1_multiplexers

Adders/Subtractors: *nb_of_adds_subs*

n-bit adder: *nb_of_n_bit_adds*

n-bit subtractor: *nb_of_n_bit_subs*

Multipliers: *nb_of_multipliers*

Logic Shifters: *nb_of_logic_shifters*

```
# Counters: nb_of_counters
  n-bit {up | down | updown} counter:  nb_of_n_bit_counters
# XORs: nb_of_xors
Cell Usage :
# BELS: nb_of_bels
  # AND...: nb_of_and...
  # OR...: nb_of_or...
  # INV: nb_of_inv
  # XOR2: nb_of_xor2
  # GND: nb_of_gnd    # VCC: nb_of_vcc
# FlipFlops/Latches: nb_of_ff_latch
  # FD...: nb_of_fd...
  # LD...: nb_of_ld...
# Tri-States: nb_of_tristates
  # BUFE: nb_of_bufe
  # BUFT: nb_of_buft
# IO Buffers: nb_of_iobuffers
  # IBUF: nb_of_ibuf
  # OBUF: nb_of_obuf
  # IOBUF: nb_of_iobuf
  # OBUFE: nb_of_obufe
  # OBUFT: nb_of_obuft    # Others: nb_of_others
```

CPLD 合成の制約

Hardware Description Language (HDL) デザインまたは制約ファイルで指定された制約 (属性) は、XST では NGC ファイルに信号プロパティとして記述されます。

CPLD 合成結果の向上

XST は、次のように CPLD フィッタ用に最適化されたネットリストを生成します。

- ・ 指定デバイスにフィット
- ・ ダウンロード プログラマブル ファイルを作成

XST での CPLD の下位レベル合成には、次が含まれます。

- ・ ロジック最小化
- ・ サブファンクション コラプス
- ・ ロジックの因数分解
- ・ ロジック分解

最適化が実行されると、ブール代数式に対応する NGC ネットリストが出力されます。CPLD フィッタでは、マクロセルの機能に最大限にフィットするようにこれらのブール代数式が再処理されます。代数式のシェーピング機能として知られる XST の特別な最適化プロセスは、次のオプションが選択されている場合に XC9500 および XC9500XL デバイ스에適用されます。

- ・ [Keep Hierarchy] オプション

No

- ・ [Optimization Effort] オプション

2 または High

- ・ [Macro Preserve] オプション

No

代数式のシェーピング機能には、クリティカル パスの最適化アルゴリズムも含まれます。このアルゴリズムは、クリティカル パスのレベル数を削減しようとします。

CPLD フィッタでは次のような特別な最適化が行われるため、マルチレベルの最適化が推奨されます。

- ・ D フリップフロップから T フリップフロップへの変換
- ・ ドモルガン ブール代数式の選択

周波数の向上

周波数はロジック レベル数によって決定されます。レベル数を削減するには、次のオプションを設定することをお勧めします。

- ・ [Optimization Effort] オプション

2 または [High] に設定すると、デザインを必要以上に複雑なものにせずにロジック レベル数を削減するため、コラプス アルゴリズムが使用されます。

- ・ [Optimization Goal] オプション

[Speed] に設定すると、レベル数の削減が優先されます。

周波数に最も影響するのは、CPLD フィッタの最適化です。CPLD フィッタのマルチレベル最適化を、**-ptersms** オプションに 20 ~ 50 までの値を増分 5 で設定して実行することをお勧めします。統計的には、30 で最高の周波数が得られます。

次のトライは、順に周波数が向上していく例を示しています。

- ・ トライ 1
- ・ トライ 2
- ・ トライ 3
- ・ トライ 4

CPU タイムは、次のトライ 1 から 4 へ増加しています。

トライ 1

[Optimization Effort] を 2、[Optimization Goal] を [Speed] に設定します。それ以外のオプションには、デフォルト値を使用します。

- ・ [Optimization Effort] :
2 または High
- ・ [Optimization Goal] オプション
Speed

トライ 2

ユーザー階層をフラット化します。階層をフラット化すると、最適化プロセスでデザインの全体像が認識され、ロジックレベルを削減しやすくなります。

- ・ [Optimization Effort] :
1/Normal または 2/High
- ・ [Optimization Goal] オプション
Speed
- ・ [Keep Hierarchy] オプション
no (チェックボックスをオフ)

トライ 3

マクロを周辺ロジックと結合します。デザインがさらにフラット化されます。

- ・ [Optimization Effort] :
1 または Normal
- ・ [Optimization Goal] オプション
Speed
- ・ [Keep Hierarchy] オプション
no (チェックボックスをオフ)
- ・ [Macro Preserve] オプション
no (チェックボックスをオフ)

トライ 4

式のシェーピング (最適化) アルゴリズムを適用します。選択するオプションは次のとおりです。

- ・ [Optimization Effort] :
2 または High
- ・ [Macro Preserve] オプション
no (チェックボックスをオフ)
- ・ [Keep Hierarchy] オプション
no (チェックボックスをオフ)

大規模デザインのフィット

デバイスのマクロセル数または積項を超過していて、選択デバイスにデザインがフィットしない場合、XST でエリア最適化を選択する必要があります。統計的には、最良のエリアが得られるのは、次のようにオプションを設定している場合です。

- ・ [Optimization Effort] : **1** (Normal) または **2** (High)
- ・ [Optimization Goal] : **Area**
- ・ その他のオプション : デフォルト値

このほかに、`-wysiwyg yes` オプションを使用できます。このオプションは、最適化プロセスでデザインを簡略化できず、積項数がデバイスで許容できる限度にほぼ達しているような場合に効果的です。ロジックレベル数を削減しようとしたために論理式が多く作成されて積項数が増加し、デザインをフィットできなくなることがあります。上記のようにオプションを設定すると、積項数が増加することはなく、デザインをフィットしやすくなります。

デザイン制約

この章では、XST のデザイン制約とその詳細について説明します。

この章は、次のセクションから構成されています。

- ・ [デザイン制約のリスト](#)
- ・ [グローバル制約とオプションの設定](#)
- ・ [VHDL 属性の構文](#)
- ・ [Verilog-2001 の属性](#)
- ・ [XST Constraint File \(XCF\)](#)
- ・ [制約の優先順位](#)
- ・ [XST 固有の制約 \(タイミング以外\)](#)
- ・ [XST コマンド ラインのみのオプション](#)

特定の XST デザイン制約の詳細は、次を参照してください。

- ・ [一般制約](#)
- ・ [HDL 制約](#)
- ・ [FPGA 制約 \(タイミング制約以外\)](#)
- ・ [CPLD 制約 \(タイミング以外\)](#)
- ・ [タイミング制約](#)
- ・ [インプリメンテーション制約](#)
- ・ [サードパーティの制約](#)

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。XST では、合成プロセスや配置配線のさまざまな処理を制御するために制約を使用できます。ほとんどの場合、合成アルゴリズムによって自動的に最適な結果が得られますが、場合によっては、最適な結果が得られないこともあります。このような場合、制約を使用することにより、別の方法で合成し直すことができます。

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する。ISE® Design Suite の [Process Properties] ダイアログ ボックスで設定するか、またはコマンドラインで run コマンドにオプションを使用して設定する。
- ・ 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する。
- ・ Verilog 属性 (推奨) またはメタ コメントとして追加する。
- ・ 制約は別の制約ファイルで指定する。

通常、グローバルな合成オプションは、ISE Design Suite の [Process Properties] ダイアログ ボックスの [Synthesis Options]、またはコマンドラインで設定します。一方、VHDL/Verilog 属性や Verilog メタ コメントはソースコードに記述するので、デザインの各エレメントに異なる制約を設定できます。

各エレメントへの設定は、グローバル設定よりも優先されます。同様に、ノード (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、ノード (またはインスタンス) の制約が優先されます。

制約を指定するには、次の一般的な規則に従う必要があります。

- ・ 信号には、複数の制約を設定できます。複数の制約を設定する場合、制約は信号が宣言され、使用されているブロック内で指定する必要があります。
- ・ 制約はエンティティ (VHDL) に適用してから、コンポーネント宣言でも適用することができます。コンポーネントに制約を適用可能なことは、これが一般的な XST の規則であるため、制約別にははっきり記載されていません。
- ・ アーキテクチャ文に制約を適用できるサードパーティの合成ツールもあります。XST でアーキテクチャに制約を設定できるのは、XST で自動的にサポートされるサードパーティ制約に対してのみです。

デザイン制約のリスト

次は、タイプ別の XST デザイン制約のリストです。

- ・ [一般制約](#)
- ・ [HDL 制約](#)
- ・ [FPGA 制約 \(タイミング制約以外\)](#)
- ・ [CPLD 制約 \(タイミング以外\)](#)
- ・ [タイミング制約](#)
- ・ [インプリメンテーション制約](#)
- ・ [サードパーティの制約](#)

一般制約

次の制約は、「一般制約」に含まれます。

- ・ 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)
- ・ 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)
- ・ 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使用 (-iuc)
- ・ Verilog の 'include ディレクトリの指定 (-vlgincdir)
- ・ Verilog 2001 の指定 (-verilog2001)
- ・ 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)

HDL 制約

次の制約は、「HDL 制約」に含まれます。

- ・ 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)
- ・ レジスタ電源投入時の値 (REGISTER_POWERUP)
- ・ 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)
- ・ 信号のエンコード方法 (SIGNAL_ENCODING)

FPGA 制約 (タイミング制約以外)

次は、XST の FPGA 制約 (タイミング制約以外) です。

- ・ 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)
- ・ デコーダの抽出 (DECODER_EXTRACT)
- ・ 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)
- ・ 論理シフタの抽出 (SHIFT_EXTRACT)
- ・ 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)
- ・ MUX スタイル (MUX_STYLE)
- ・ グローバル クロック バッファ数 (-bufg)
- ・ リージョン クロック バッファ数 (-bufr)
- ・ 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)

- ・ プライオリティ エンコーダの抽出 (PRIORITY_EXTRACT)
- ・ 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)
- ・ XOR コラプス (XOR_COLLAPSE)
- ・ スライス (LUT-FF ペア) 使用率 (SLICE_UTILIZATION_RATIO)
- ・ スライス (LUT-FF ペア) 使用率の許容範囲 (SLICE_UTILIZATION_RATIO_MAXMARGIN)
- ・ 単一 LUT へのエンティティのマップ (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)
- ・ DSP48 の使用 (USE_DSP48)

CPLD 制約 (タイミング以外)

次の制約は、「CPLD 制約 (タイミング以外)」に含まれます。

- ・ クロック イネーブル (-pld_ce)
- ・ データ ゲート (DATA_GATE)
- ・ マクロの保持 (-pld_mp)
- ・ 削減なし (NOREDUCE)
- ・ WYSIWYG (-wysiwyg)
- ・ XOR の保持 (-pld_xp)

タイミング制約

次の制約は、「[タイミング制約](#)」に含まれます。

- ・ [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 \(CLOCK_SIGNAL\)](#)
- ・ [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 制約 \(FROM-TO\)](#)
- ・ [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 \(TIG\)](#)

インプリメンテーション制約

次の制約は、「[インプリメンテーション制約](#)」に含まれます。

- ・ [RLOC](#)
- ・ [NOREDUCE](#)
- ・ [PWR_MODE](#)

サードパーティの制約

「[サードパーティの制約](#)」では、サードパーティ制約の説明とそれに対応する XST 制約について説明されます。

グローバル制約とオプションの設定

このセクションでは、ISE® Design Suite の [Process Properties] ダイアログ ボックスでグローバル制約およびオプションを設定する方法を説明します。

FPGA デバイス、CPLD デバイス、VHDL、Verilog などでも一般的に適用される制約についての詳細は、『[制約ガイド](#)』を参照してください。

チェック ボックス付きのフィールド以外の値フィールドには、プルダウン メニューの矢印ボタンや参照ボタンが付いています。ボックスをクリックするまでは、矢印ボタンは画面上に表示されません。

合成オプション設定

ISE® Design Suite から Hardware Description Language (HDL) 合成オプションを設定するには

1. [Sources] ウィンドウでソース ファイルを選択します。
2. [Processes] ウィンドウで [Synthesize - XST] 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. デバイス ファミリ (FPGA または CPLD) の選択によって、表示されるダイアログ ボックスは異なります。
6. 次の合成オプションのいずれかを選択します。
 - ・ 最適化方法の指定 (OPT_MODE)
 - ・ 最適化エフォート レベル (OPT_LEVEL)
 - ・ 合成制約ファイルの使用 (-iuc)
 - ・ 合成制約ファイルの指定 (-uc)
 - ・ ライブラリの検索順 (-lso)
 - ・ グローバルな最適化目標 (-glob_opt)
 - ・ RTL 回路図の生成 (-rtlview)
 - ・ タイミング制約の書き込み (-write_timing_constraints)
 - ・ Verilog 2001 の指定 (-verilog2001)

次のオプションを表示するには、[Process Properties] ダイアログ ボックスで [Property display level] を [Advanced] に設定します。

- ・ 階層の維持 (KEEP_HIERARCHY)
- ・ コアの検索ディレクトリ (-sd)
- ・ クロス クロック解析 (-cross_clock_analysis)
- ・ 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)
- ・ 作業ディレクトリの指定 (-xsthdpdir)
- ・ HDL ライブラリ マップ ファイル (-xsthdpini)
- ・ Verilog の 'include ディレクトリの指定 (-vlgincdir)
- ・ スライス (LUT-FF ペア) 使用率 (SLICE_UTILIZATION_RATIO)

Hardware Description Language (HDL) オプションの設定

FPGA と CPLD の Hardware Description Language (HDL) オプションはユーザーが設定できます。

FPGA デバイスの Hardware Description Language (HDL) オプションの設定

FPGA デバイスの Hardware Description Language (HDL) オプションは、ISE® Design Suite で [Synthesize - XST] プロセスを右クリックし、[Process Properties] をクリックして表示される [Process Properties] ダイアログ ボックスの [Hardware Description Language (HDL) Options] ページから設定できます。

FPGA デバイスの場合は、次の HDL オプションを設定できます。

- ・ FSM エンコード方法の指定 (FSM_ENCODING)
- ・ セーフ インプリメンテーション (SAFE_IMPLEMENTATION)
- ・ Case 文のインプリメンテーション形式 (-vlgcase)
- ・ FSM スタイル (FSM_STYLE)

FSM スタイル オプションを表示するには、[Process Properties] ダイアログ ボックスで [Property display level] を [Advanced] に設定します。

- ・ 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)
- ・ MUX の抽出 (MUX_EXTRACT)
- ・ MUX スタイル (MUX_STYLE)
- ・ デコーダの抽出 (DECODER_EXTRACT)
- ・ プライオリティ エンコーダの抽出 (PRIORITY_EXTRACT)
- ・ シフトレジスタの抽出 (SHREG_EXTRACT)
- ・ 論理シフタの抽出 (SHIFT_EXTRACT)
- ・ XOR コラプス (XOR_COLLAPSE)
- ・ リソース共有 (RESOURCE_SHARING)
- ・ 乗算器スタイル (MULT_STYLE)

新しいデバイスの場合、この [Multiplier Style] プロパティの代わりに 次のプロパティが表示されます。

- ・ DSP48 の使用 (Virtex®-4 デバイス)
- ・ DSP ブロックの使用 (Virtex-5 および Spartan®-3A DSP デバイス)
- ・ DSP48 の使用 (USE_DSP48)

CPLD デバイスの Hardware Description Language (HDL) オプションの設定

CPLD デバイスの Hardware Description Language (HDL) オプションは、ISE Design Suite で [Synthesize - XST] プロセスを右クリックし、[Process Properties] をクリックして表示される [Process Properties] ダイアログ ボックスの [Hardware Description Language (HDL) Options] ページから設定できます。

CPLD デバイスの場合は、次の HDL オプションを設定できます。

- ・ FSM エンコード方法の指定 (FSM_ENCODING)
- ・ セーフ インプリメンテーション (SAFE_IMPLEMENTATION)
- ・ Case 文のインプリメンテーション形式 (-vlgcase)
- ・ MUX の抽出 (MUX_EXTRACT)
- ・ リソース共有 (RESOURCE_SHARING)

ザイリンクス特有オプションの設定

次のデバイスには、ザイリンクス特有のオプションを設定できます。

- ・ FPGA デバイス
- ・ CPLD デバイス

FPGA デバイスのザイリンクス特有オプションの設定

FPGA デバイスのザイリンクス特有のオプションは、ISE® Design Suite で [Synthesize - XST] プロセスを右クリックし、[Properties] をクリックして表示される [Process Properties] ダイアログ ボックスの [Xilinx Specific Options] ページから設定できます。

FPGA デバイスの場合は、次のザイリンクス特有オプションを設定できます。

- ・ I/O バッファの追加 (-iobuf)
- ・ LUT の結合
- ・ 最大ファンアウト数 (MAX_FANOUT)
- ・ レジスタの複製 (REGISTER_DUPLICATION)
- ・ 制御セットの削減 (REDUCE_CONTROL_SETS)
- ・ 等価レジスタの削除 (EQUIVALENT_REGISTER_REMOVAL)
- ・ 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)
- ・ トライステートからロジックへの変換 (TRISTATE2LOGIC)

これは、内部トライステートリソースを含むデバイスの場合にのみ表示されます。

- ・ 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)

次のオプションを表示するには、[Process Properties] ダイアログ ボックスで [Property display level] を [Advanced] に設定します。

- ・ グローバル クロック バッファ数 (-bufg)
- ・ リージョン クロック バッファ数 (-bufr)

CPLD デバイスのザイリンクス特有オプションの設定

CPLD デバイスのザイリンクス特有のオプションは、ISE® Design Suite で [Synthesize - XST] プロセスを右クリックし、[Properties] をクリックして表示される [Process Properties] ダイアログ ボックスの [Xilinx Specific Options] ページから設定できます。

CPLD デバイスの場合は、次のザイリンクス特有オプションを設定できます。

- ・ I/O バッファの追加 (-iobuf)
- ・ 等価レジスタの削除 (EQUIVALENT_REGISTER_REMOVAL)
- ・ クロック イネーブル (-pld_ce)
- ・ マクロの保持 (-pld_mp)
- ・ XOR の保持 (-pld_xp)
- ・ WYSIWYG (-wysiwyg)

その他の XST コマンド ライン オプションの設定

ISE® Design Suite の [Process Properties] ダイアログ ボックスの [Synthesis Options] ページにある [Other XST Command Line Options] プロパティで、その他の XST のコマンド ライン オプションを指定できます。これは、アドバンス プロパティです。構文については、「[コマンド ライン モード](#)」を参照してください。複数のオプションを入力する場合は、スペースで区切ります。

このプロパティは、[Process Properties] ダイアログ ボックスにリストされていないオプションを設定するためのものですが、どのオプションでも入力できます。ただし、ほかのプロパティで既に指定されているオプションがあれば、そちらが優先されます。オプションの入力方法が不正だったり、認識されないオプションを入力すると、次のようなエラー メッセージが表示され、XST での処理が停止します。

```
ERROR:Xst:1363 - Option "-verilog2002" is not available for command run.
```

カスタム コンパイル ファイル リスト

[Custom Compile File List] プロパティを使用すると、XST でソース ファイルを処理する順序を変更できます。このプロパティで、ライブラリおよびデザイン ファイルの処理順を指定したコンパイル リスト ファイルを指定します。このプロパティでファイルを指定しない場合は、自動的に生成されたリストが使用されます。

カスタム コンパイル リスト ファイルには、すべてのデザイン ファイルおよび関連するライブラリをコンパイルする順にリストします。ファイルとライブラリの組を 1 行に 1 つずつリストし、ファイルとライブラリはセミコロンで区切ります。フォーマットは次のとおりです。

```
library_name;file_name [library_name;file_name] ...
```

次にコマンド例を示します。

```
work;stopwatch.vhd  
work;statmach.vhd  
...
```

このプロパティは、[Simulation Properties] ページの [Custom Compile File List] プロパティとは関係ありません。合成とシミュレーションでは異なるコンパイル リスト ファイルが使用されます。

VHDL 属性の構文

VHDL コードの場合、制約は VHDL 属性を使用して記述できます。次の構文で属性を宣言してから、属性を使用します。

```
attribute AttributeName : Type ;
```

VHDL 属性の構文例 1

```
attribute RLOC : string ;
```


属性タイプでは、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。エンティティで宣言した場合、その属性はエンティティおよびアーキテクチャ本体の両方で使用できます。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 属性を宣言後、次のように指定します。

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

VHDL 属性の構文例 2

```
attribute RLOC of u123 : 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、エンティティ (entity)、コンポーネント (component)、ラベル (label)、信号 (signal)、変数 (variable)、タイプ (type) です。

制約を指定するには、次の一般的な規則に従う必要があります。

- ・ 制約はエンティティ (VHDL) に適用してから、コンポーネント宣言でも適用することができます。コンポーネントに制約を適用可能なことは、これが一般的な XST の規則であるため、制約別にははっきり記載されていません。
- ・ アーキテクチャ文に制約を適用できるサードパーティの合成ツールもあります。XST でアーキテクチャに制約を設定できるのは、XST で自動的にサポートされるサードパーティ制約に対してのみです。

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、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します。
 - module
 - instance
 - net
- ・ 次の合成制約を設定してください。
 - フル ケース (FULL_CASE)
 - パラレル ケース (PARALLEL_CASE)

Verilog-2001 属性の構文

Verilog-2001 の属性は、(*) で囲む必要があり、次のように記述されます。

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

この場合、それぞれ次を表します。

- ・ 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前に記述する必要があります。
- ・ 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。
- ・ デフォルトは 1 です。
- ・ (* attribute_name *) は (* attribute_name = "1" *) と同じです。

Verilog-2001 属性の構文例 1

```
(* clock_buffer = "IBUFG" *) input CLK;
```

Verilog-2001 属性の構文例 2

```
(* INIT = "0000" *) reg [3:0] d_out;
```

Verilog-2001 属性の構文例 3

```
always@(current_state or reset) begin (* parallel_case *) (* full_case *) case  
(current_state) ...
```

Verilog-2001 属性の構文例 4

```
(* mult_style = "pipe_lut" *) MULT my_mult (a, b, c);
```

Verilog-2001 の制限

次の Verilog-2001 属性は、サポートされていません。

- ・ 信号の宣言
- ・ ステートメント
- ・ ポートの接続
- ・ 論理演算子

Verilog-2001 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 形式がよく使用されますが、Verilog でもメタ コメントがサポートされています。次の構文を使用してください。

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

Verilog-2001 のメタ コメントの例

```
// 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。

- ・ [パラレル ケース \(PARALLEL_CASE\)](#)
- ・ [フル ケース \(FULL_CASE\)](#)
- ・ [変換なし \(TRANSLATE_OFF\)](#) と [変換あり \(TRANSLATE_ON\)](#)

詳細は、「[Verilog の属性とメタ コメント](#)」を参照してください。

XST Constraint File (XCF)

XST 制約は、XST Constraint File (XCF) で指定できます。制約ファイルの拡張子は .xcf です。ISE® Design Suite で XCF を指定する方法については、ISE Design Suite ヘルプを参照してください。

コマンドラインで XCF を指定するには、run コマンドで[合成制約ファイル \(-uc\)](#) オプションを使用します。コマンドラインからの XST の起動方法と run コマンドについては、「[XST コマンドライン モード](#)」を参照してください。

XST Constraint File (XCF) 構文とその使用

XST Constraint File (XCF) 構文を使用すると、特定の制約を次のいずれかに指定できます。

- ・ デバイス全体 (グローバル)
- ・ 特定モジュール

この構文は、ネットまたはインスタンスに制約を設定するという点では UCF 構文と同じですが、さらに詳細に指定することで階層の特定レベルに制約を適用できます。キーワード **MODEL** を、制約を適用するエンティティ/モジュールの定義に使用できます。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。

通常は、まず ISE® Design Suite の [Process Properties] ダイアログ ボックスかコマンドラインで指定してから、XCF ファイルで例外的な制約を指定します。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 ;
```

XST Constraint File (XCF) の例 1

```
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エートすると、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。

```
BEGIN MODEL entityname INST instancename constraintname = constraintvalue ; NET
signalname constraintname = constraintvalue ; END;
```

XST Constraint File (XCF) の例 2

```
BEGIN MODEL crc32
INST stopwatch opt_mode = area ;
INST U2 ram_style = block ;
NET myclock clock_buffer = true ;
NET data_in iob = true ;
END;
```

XST で適用できる合成制約のリストは、「[XST 固有のオプション \(タイミング以外\)](#)」を参照してください。

ネイティブ UCF 制約とそれ以外の UCF 制約

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 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)
- ・ 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)
- ・ From-To 制約 (FROM-TO)

これらの制約を **BEGIN MODEL...** **END** 構文の中に使用すると、XST でエラーが発生します。

ネイティブ以外の User Constraints File (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。

- ・ 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)

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。詳細は、[階層区切り文字の指定 \(-hierarchy_separator\)](#) を参照してください。

XST Constraint File (XCF) の構文の制限

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

- ・ 今リリースでは、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 UCF 制約がサポートされているわけではありません。詳細については、『[制約ガイド](#)』を参照してください。

制約の優先順位

制約の優先順位は、その制約が含まれるファイルによって異なります。デザイン フローの前半で使用されるファイルの制約よりも、フローの後半で使用されるファイルの制約の方が優先されます。優先順位は次のとおりです。

1. 合成制約ファイル
2. Hardware Description Language (HDL) ファイル
3. ISE® Design Suite の [Process Properties] またはコマンドライン

XST 固有の制約 (タイミング以外)

次の表に示す項目は、次のとおりです。

- ・ 各制約に使用できる値
- ・ 適用できるオブジェクトの種類
- ・ 使用制限

多くの場合、制約はエンティティまたはモデル全体にグローバルに適用するか、または個々の信号、ネット、またはインスタンスにローカルに適用できます。

XST 固有の制約 (タイミング以外)

制約名	制約 値	VHDL ターゲット	Verilog ターゲッ ト	XCF ターゲット	コマンドライン	コマンド値
BoxType	primitive	entity	module	model	なし	なし
	black_box	inst	inst	inst (in model)		
	user_black_box					
Map Logic on BRAM	yes no (チェックボッ クスをオフ)	entity	module	model	なし	なし
Buffer Type	bufgdl	signal	signal	net (in model)	なし	なし
	ibufg					
	bufg					
	bufgp					
	ibuf					
	bufr					
	none					
Clock Signal	yes	primary	primary	net (in model)	-bufgce	yes
	no	clock	clock			no
		signal	signal			デフォルト : no
Clock Signal	yes	clock	clock	clock	なし	なし
	no	signal	signal	signal		
				net (in model)		
Decoder Extraction	yes	entity	entity	model	-decoder _extract	yes
	no	signal	signal	net (in model)		no デフォルト : yes

制約名	制約 値	VHDL ターゲット	Verilog ターゲッ ト	XCF ターゲット	コマンド ライン	コマンド値
Enumerated Encoding	string containing space-separated binary codes	type	signal	net (in model)	なし	なし
Equivalent Register Removal	yes no	entity signal	module signal	model net (in model)	-equivalent _register _removal	yes no デフォルト : yes
FSM Encoding Algorithm	auto one-hot compact sequential gray johnson speed1 user	entity signal	module signal	model net (in model)	-fsm _encoding	auto one-hot compact sequential gray johnson speed1 user デフォルト : auto
Automatic FSM Extraction	yes no	entity signal	module signal	model net (in model)	-fsm _extract	yes no デフォルト : yes
FSM Style	lut bram	entity signal	module signal	model net (in model)	-fsm _style	lut bram デフォルト : lut
Full Case	なし	なし	case statement	なし	なし	なし
Pack I/O Registers Into IOBs	true false auto	signal instance	signal instance	net (in model) inst (in model)	-iob	true false auto デフォルト : auto
I/O Standard	string 詳細は、『 制約 ガイド 』を参照し てください。	signal instance	signal instance	net (in model) inst (in model)	なし	なし

制約名	制約 値	VHDL ターゲット	Verilog ターゲッ ト	XCF ターゲット	コマンドライン	コマンド値
Keep	true false soft	signal	signal	net (in model)	なし	なし
Keep Hierarchy	yes no soft	entity	module	model	-keep _hierarchy	yes no soft デフォルト (FPGA) : no デフォルト (CPLD) : yes
LOC	string	signal (primary IO) instance	signal (primary IO) instance	net (in model) inst (in model)	なし	なし
Map Entity on a Single LUT	yes no	entity architecture	module	model	なし	なし
Max Fanout	integer	entity signal	module signal	model net (in model)	-max _fanout	integer デフォルト : 詳 細な説明を参照 してください。
Move First Stage	yes no	entity primary clock signal	module primary clock signal	model primary clock signal net (in model)	-move _first _stage	yes no デフォルト : yes
Move Last Stage	yes no	entity primary clock signal	module primary clock signal	model primary clock signal net (in model)	-move _last _stage	yes no デフォルト : yes

制約名	制約 値	VHDL ターゲット	Verilog ターゲッ ト	XCF ターゲット	コマンド ライン	コマンド値
Multiplier Style	auto block pipe_block kcm csd lut pipe_lut	entity signal	module signal	model net (in model)	-mult _style	auto block pipe_block kcm csd lut pipe_lut デフォルト : auto
Mux Extraction	yes no force	entity signal	module signal	model net (in model)	-mux _extract	yes no force デフォルト : yes
Mux Style	auto muxf muxcy	entity signal	module signal	model net (in model)	-mux _style	auto muxf muxcy デフォルト : auto
No Reduce	yes no	signal	signal	net (in model)	なし	なし
Optimization Effort	1 2	entity	module	model	-opt _level	1 2 デフォルト : 1
Optimization Goal	speed area	entity	module	model	-opt _mode	speed area デフォルト : speed
Optimize Instantiated Primitives	yes no	entity instance	module instance	model instance (in model)	-optimize _primitives	yes no デフォルト : no
Parallel Case	なし	なし	case statement	なし	なし	なし

制約名	制約 値	VHDL ターゲット	Verilog ターゲッ ト	XCF ターゲット	コマンド ライン	コマンド値
Power Reduction	yes no	entity	module	model	-power	yes no デフォルト : no
Priority Encoder Extraction	yes no force	entity signal	module signal	model net (in model)	-priority _extract	yes no force デフォルト : yes
RAM Extraction	yes no	entity signal	module signal	model net (in model)	-ram _extract	yes no デフォルト : yes
RAM Style	auto block distributed pipe_distributed block_power1 block_power2	entity signal	module signal	model net (in model)	-ram _style	auto block distributed デフォルト : auto
Read Cores	yes no optimize	entity component	module label	model inst (in model)	-read _cores	yes no optimize デフォルト : yes
Register Balancing	yes no forward backward	entity signal FF instance name	module signal FF instance name primary clock signal	model net (in model)inst (in model)	-register _balancing	yes no forward backward デフォルト : no
Register Duplication	yes no	entity signal	module	model net (in model)	-register _duplication	yes no デフォルト : yes
Register Power Up	string	type	signal	net (in model)	なし	なし

制約名	制約 値	VHDL ターゲット	Verilog ターゲッ ト	XCF ターゲット	コマンド ライン	コマンド値
Resource Sharing	yes no	entity signal	module signal	model net (in model)	-resource _sharing	yes no デフォルト : yes
ROM Extraction	yes no	entity signal	module signal	model net (in model)	-rom _extract	yes no デフォルト : yes
ROM Style	auto block distributed	entity signal	module signal	model net (in model)	-rom _style	auto block distributed デフォルト : auto
Save	yes no	signal inst of primitive	signal inst of primitive	net (in model) inst of primitive (in model)	なし	なし
Safe Implementation	yes no	entity signal	module signal	model net (in model)	-safe _implementation	yes no デフォルト : no
Safe Recovery State	string	signal	signal	net (in model)	なし	なし
Logical Shifter Extraction	yes no	entity signal	module signal	model net (in model)	-shift _extract	yes no デフォルト : yes
Shift Register Extraction	yes no	entity signal	module signal	model net (in model)	-shreg extract	yes no デフォルト : yes
Signal Encoding	auto one-hot user	entity signal	module signal	model net (in model)	-signal _encoding	auto one-hot user デフォルト : auto

制約名	制約 値	VHDL ターゲット	Verilog ターゲッ ト	XCF ターゲット	コマンド ライン	コマンド値
Slice Utilization Ratio	integer (range -1 to 100) integer% (range -1 to 100) integer#	entity	module	model	-slice _utilization _ratio	integer (range -1 to 100) integer% (range -1 to 100) integer# デフォルト : 100
Slice Utilization Ratio Delta	integer (range 0 to 100) integer% (range 0 to 100) integer#	entity	module	model	-slice _utilization _ratio _maxmargin	integer (range 0 to 100) integer% (range 0 to 100) integer# デフォルト : 0
Translate Off Translate On	なし	local no target	local no target	なし	なし	なし
Convert Tristates to Logic	yes no	entity signal	modulesignal	model net (in model)	-tristate2logic	yes no デフォルト : yes
Use Carry Chain	yes no	entity signal	module signal	model net (in model)	-use _carry _chain	yes no デフォルト : yes
Use Clock Enable	auto yes no	entity signal FF instance name	module signal FF instance name	model net (in model) inst (in model)	-use _clock _enable	auto yes no デフォルト : auto
Use DSP48	auto yes no	entity signal	module signal	model net (in model)	-use _dsp48	auto yes no デフォルト : auto

制約名	制約 値	VHDL ターゲット	Verilog ターゲッ ト	XCF ターゲット	コマンドライン	コマンド値
Use Synchronous Reset	auto yes no	entity signal FF instance name	module signal FF instance name	model net (in model) inst (in model)	-use _sync _reset	auto yes no デフォルト : auto
Use Synchronous Set	auto yes no	entity signal FF instance name	module signal FF instance name	model net (in model) inst (in model)	-use _sync _set	auto yes no デフォルト : auto
XOR Collapsing	yes no	entity signal	module signal	model net (in model)	-xor _collapse	yes no デフォルト : yes

XST コマンド ラインのみのオプション

XST 固有の制約 (タイミング以外) : XST コマンド ラインのみ

制約名	コマンドライン	コマンド値
VHDL Top Level Architecture	-arch	architecture_name デフォルト : なし
Asynchronous to Synchronous	-async_to_sync	yes no デフォルト : no
Automatic BRAM Packing	-auto_bram_packing	yes no デフォルト : no
BRAM Utilization Ratio (BRAM_UTILIZATION_RATIO)	-bram_utilization_ ratio	integer (range -1 to 100) integer% (range -1 to 100) integer# デフォルト : 100

制約名	コマンドライン	コマンド値
Maximum Global Clock Buffers	-bufg	整数 デフォルト: ターゲット デバイスの最大バッファ数
Maximum Regional Clock Buffers	-bufr	整数 デフォルト: ターゲット デバイスの最大バッファ数
Bus Delimiter	-bus_delimiter	< > [] { } () デフォルト: < >
Case	-case	upper lower maintain デフォルト: maintain
Verilog Macros	-define	{name = value} デフォルト: なし
DSP Utilization Ratio (DSP_UTILIZATION_RATIO)	-dsp_utilization_ratio	integer (range -1 to 100) integer% (range -1 to 100) integer# デフォルト: 100
Duplication suffix	-duplication_suffix	string%dstring デフォルト: _%d
VHDL Top-Level block (以前の VHDL プロジェクト フォーマット (-ifmt VHDL) でのみ使用可能。合成する最上位レベルのブロックを指定するには、プロジェクト フォーマット (-ifmt mixed) か -top オプションを使用してください)	-ent	entity_name デフォルト: なし
Generics	-generics	{name = value} デフォルト: なし
HDL File Compilation Order	-hdl_compilation_order	auto user デフォルト: auto

制約名	コマンドライン	コマンド値
Hierarchy Separator	-hierarchy_separator	- / デフォルト : /
Input Format	-ifmt	mixed vhdl verilog デフォルト : mixed
Input/Project File Name	-ifn	ファイル名 デフォルト : なし
Add I/O Buffers	-iobuf	yes no デフォルト : yes
Ignore User Constraints	-iuc	yes no デフォルト : no
Library Search Order	-lso	file_name.lso デフォルト : なし
LUT Combining	-lc	auto area off デフォルト : off
Netlist Hierarchy	-netlist_hierarchy	as_optimized rebuilt デフォルト : as_optimized
Output File Format	-ofmt	NGC デフォルト : NGC
Output File Name	-ofn	ファイル名 デフォルト : なし
Target Device	-p	part-package-speed (例 : xc5vfx30t-f324-2) デフォルト : なし

制約名	コマンドライン	コマンド値
Clock Enable	-pld_ce	yes no デフォルト : yes
Macro Preserve	-pld_mp	yes no デフォルト : yes
XOR Preserve	-pld_xp	yes no デフォルト : yes
Reduce Control Sets	-reduce_control_sets	auto no デフォルト : no
Generate RTL Schematic	-rtlview	yes no only デフォルト : no
Cores Search Directories	-sd	ディレクトリ デフォルト : なし
Slice Packing	-slice_packing	yes no デフォルト : yes
Top Level Block	-top	block_name デフォルト : なし
Synthesis Constraints File	-uc	file_name.xcf デフォルト : なし
Verilog 2001	-verilog2001	yes no デフォルト : yes
Case Implementation Style	-vlgcase	full parallel full-parallel デフォルト : なし

制約名	コマンド ライン	コマンド値
Verilog Include Directories	-vlgincdir	ディレクトリ デフォルト : なし
Work Library	-work_lib	ディレクトリ デフォルト : work
wysiwyg	-wysiwyg	yes no デフォルト : no
Work Directory	-xsthdpdir	ディレクトリ デフォルト : ./xst
HDL Library Mapping File	-xsthdpini	file_name.ini デフォルト : なし

XST タイミング オプション

XST のタイミング オプションは、次のように指定できます。

- ・ ISE® Design Suite のプロセス プロパティ
- ・ コマンド ライン
- ・ XST Constraint File (XCF)

XST タイミング オプション : [Process Properties] またはコマンド ライン

次の表では、コマンド ラインまたはISE® Design Suite の [Process Properties] ダイアログ ボックスからのみ起動可能な XST でサポートされるタイミング制約を示しています。

コマンド ラインまたは[Process Properties] ダイアログ ボックスでのみサポートされる XST タイミング制約

オプション	プロセス プロパティ (ISE Design Suite)	値
glob_opt	Global Optimization Goal	allclocknetsinpad _to_outpadoffset _in_beforeoffset _out_aftermax _delay デフォルト : allclocknets
cross_clock_analysis	Cross Clock Analysis	yes no デフォルト : no
write_timing_constraints	Write Timing Constraints	yes no

オプション	プロセス プロパティ (ISE Design Suite)	値
		デフォルト : no

XST タイミング オプション : XST Constraint File (XCF)

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 制約 \(FROM-TO\)](#)
- ・ [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 \(TIG\)](#)
- ・ [タイムスペック \(TIMESPEC\)](#)
- ・ [タイムスペック識別子 \(TSidentifier\)](#)

これらのタイミング制約により合成最適化が影響を受けます。制約を配置配線まで渡すには、[\[Write Timing Constraints\]](#) 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

次の一般的な制約は FPGA デバイス、CPLD デバイス、VHDL、Verilog に適用されます。多くの制約は、ISE® Design Suite の [Process Properties] ダイアログ ボックスの [Synthesis Options] で設定できます。

- ・ 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)
- ・ 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)
- ・ 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使用 (-iuc)
- ・ Verilog の 'include ディレクトリの指定 (-vlgincdir)
- ・ Verilog 2001 の指定 (-verilog2001)
- ・ 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)

-iobuf (I/O バッファの追加)

-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 バッファを XST で挿入しない場合は、この -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。

この値は、ISE® Design Suite の [Process Properties] ダイアログ ボックスの [Xilinx Specific Options] ページにある [Add I/O Buffers] でも設定できます。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体に適用されます。

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

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

次の値が使用できます。

- ・ **yes** (デフォルト) : IBUF/OBUF プリミティブが生成され、最上位レベル モジュールの I/O ポートに接続されます。
- ・ **no** : IBUF および OBUF プリミティブは生成されません。大型デザインの場合に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 が呼び出される際には、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。
- ・ **true**
- ・ **false**
- ・ **soft**

構文例

```
xst run -iobuf yes
```

このコマンドライン例では、I/O バッファがデザインの最上位レベルのモジュールに追加されます。

BOX_TYPE (ボックス タイプ)

ボックス タイプ (BOX_TYPE) は、合成制約です。

この制約には次のような値が使用できます。

- ・ **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 XST Constraint File (XCF) のためにインプリメントされた制約で、VHDL のように BOX_TYPE 制約をコンポーネントに適用できます。

アーキテクチャ サポート

アーキテクチャに依存しません。

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
- ・ XST Constraint File (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 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 構文例

次をインスタンス化エーションの直前に入力します。

```
(* box_type = "{primitive | black_box | user_black_box}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" box_type="{primitive | black_box | user_black_box}";
```

XST Constraint File (XCF) 構文例 2

```
BEGIN MODEL "entity_name"
```

```
INST "instance_name"
```

```
box_type="{primitive | black_box | user_black_box}";
```

```
END;
```

-bus_delimiter (バスの区切り文字指定)

-bus_delimiter (バスの区切り文字指定) で指定した文字が出力されるネットリストで使用されます。

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 [Bus Delimiter] でも設定できます。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

構文に適用されます。

適用ルール

ありません。

構文

```
-bus_delimiter {<> | [ ] | { } | ( ) }
```

The default delimiter is <>.

構文例

```
xst run -bus_delimiter [ ]
```

この例では、バス区切り文字が角かっこ ([]) にグローバルに設定されます。

-case (大文字/小文字の指定)

-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。

ISE® Design Suite の [Process Properties] ダイアログ ボックスの [Synthesis Options] ページにある [Case] で設定できます。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

構文に適用されます。

適用ルール

ありません。

構文

```
-case {upper | lower | maintain }
```

デフォルト値は maintain です。

構文例

```
xst run -case upper
```

この例では、大文字にグローバルに設定されています。

-vlgcase (Case 文のインプリメンテーション形式)

-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。

-vlgcase 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。値には、次のいずれかを指定します。

- ・ **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。
- ・ このオプションを指定しない場合は (Project Navigator では [None])、case 文のビヘイビアがそのままインプリメントされます。

ISE® Design Suite の [Process Properties] ダイアログ ボックスの [HDL Options] ページにある [Case Implementation Style] でも設定できます。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

詳細は、次を参照してください。

- ・ [マルチプレクサの Hardware Description Language \(HDL\) コーディング手法](#)
- ・ [フル ケース \(FULL_CASE\)](#)
- ・ [パラレル ケース \(PARALLEL_CASE\)](#)

ISE® Design Suite の [Process Properties] ダイアログ ボックスの [HDL Options] ページにある [Case Implementation Style] でも設定できます。

構文

```
-vlgcase {full |parallel |full-parallel }
```

By default, there is no value.

構文例

```
xst run -vlgcase full
```

-define (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るので、これにより、IP コアの生成やテストベンチフローなどの Hardware Description Language (HDL) ソース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。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。

これは、Verilog デザインにのみ使用できます。

ISE® Design Suite で Verilog マクロを定義するには

1. [Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスを表示して、[Synthesis Options] をクリックします。
2. [Process Properties] ダイアログ ボックスで [Property display level] → [Advanced] をクリックします。
3. [Verilog Macros] プロパティを設定します。

マクロを指定する場合は、{} は使用しないでください。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

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

```
-define {name[=value] name[=value]}
```

- ・ *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 {}
```

メモ :

- ・ 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 Virtex4"}
```

構文例

```
xst run -define macro1=Xilinx macro2="Xilinx Virtex4"
```

macro1 および macro2 という名前の 2 つのマクロが定義されます。

-duplication_suffix (複製接尾語の設定)

-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。デフォルトでは、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` で、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`

`-duplication_suffix` を使用すると、元の名前に追加された文字列を変更できます。

ISE® Design Suite からこの値を設定するには

1. [Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスを表示して、[Synthesis Options] をクリックします。
2. [Process Properties] ダイアログ ボックスで [Property display level] → [Advanced] をクリックします。
3. [Other XST Command Line Options] プロパティを設定します。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能なエレメント

ファイルに適用されます。

適用ルール

ありません。

構文

`-duplication_suffix string%dstring`

デフォルトは `%d` です。

構文例 1

`xst run -duplication_suffix _dupreg_%d`

このコマンドを指定すると、フリップフロップ名が `my_ff` で、同じフリップフロップが 3 度複製される場合、名前はそれぞれ次のようになります。

- ・ `my_ff_dupreg_1`
- ・ `my_ff_dupreg_2`
- ・ `my_ff_dupreg_3`

構文例 2

`xst run -duplication_suffix _dup_%d_reg`

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。このコマンドを指定すると、フリップフロップ名が `my_ff` で、同じフリップフロップが 3 度複製される場合、名前はそれぞれ次のようになります。

- ・ `my_ff_dup_1_reg`
- ・ `my_ff_dup_2_reg`
- ・ `my_ff_dup_3_reg`

FULL_CASE (フル ケース)

FULL_CASE は、Verilog デザインにのみ使用できます。FULL_CASE は、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、「[マルチプレクサの Hardware Description Language \(HDL\) 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

アーキテクチャに依存しません。

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 構文例

構文は次のとおりです。

```
(* full_case *)
```

FULL_CASE にはターゲット参照が含まれないため、セレクトのすぐ後ろに属性を記述します。

```
(* full_case *)
casex select
4'b1xxx: res = data1;
4'b01xx: res = data2;
4'b001x: res = data3;
4'b000x: res = data4;
endcase
```

FULL_CASE 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
```

FULL_CASE にはターゲット参照が含まれないため、セレクトのすぐ後ろにメタ コメントを記述します。

```
casex select // synthesis full_case
4'b1xxx: res = data1;
4'b01xx: res = data2;
4'b001x: res = data3;
4'b000x: res = data4;
endcase
```

XST コマンドライン

XST コマンドラインで次のように定義します。

```
-vlgcase [full|parallel|full-parallel]
```

ISE Design Suite からの設定

メモ : Verilog ファイルのみで設定できます。

ISE® Design Suite で次のように定義します。

[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。

-rtlview (RTL 回路図の生成)

-rtlview を使用すると、デザインの Register Transfer Level (RTL) 構造を表すネットリスト ファイルを生成できます。このネットリストは、RTL and Technology Viewers で表示できます。値には、次のいずれかを指定します。

- ・ **yes** : RTL ビューが生成されます。
- ・ **no** : RTL ビューは生成されません。
- ・ **only** : RTL ビューが生成されると合成が停止します。

RTL 表示を含むファイルの拡張子は .ngr です。

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 [Generate RTL Schematic] でも設定できます。

アーキテクチャ サポート

アーキテクチャに依存しません。

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

-rtlview {yes|no|only}

デフォルトでは no に設定されています。

構文例

-rtlview yes

XST でデザインの RTL 構造を記述するネットリスト ファイルが生成されます。

-generics (ジェネリック)

-generics を使用すると、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。これにより、IP コアの生成やテストフローなどの HDL ソース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。定義された値が VHDL または Verilog 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、VHDL または Verilog 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 VHDL または Verilog 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。

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 [Generics, Parameters] でも設定できます。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

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

```
xst run -generics {name=value name=value ...}
```

name: 最上位レベルのデザイン ブロックのジェネリックまたはパラメータの名前を示します。

value: 最上位レベルのデザイン ブロックのジェネリックまたはパラメータの値を示します。

デフォルトでは何も定義されていません。

```
-generics {}
```

次の規則に従ってください。

- ・ {} 内に値を挿入します。
- ・ 各値は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**
- ・ 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。

構文例

```
-generics {company="Xilinx" width=5 init_vector=b100101}
```

このコマンドは、company を Xilinx、幅を 5、init_vector を b100101 に設定します。

-hierarchy_separator (階層区切り文字の指定)

-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。

ISE® Design Suite から階層区切り文字を指定するには

1. [Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスを表示して、[Synthesis Options] をクリックします。
2. [Process Properties] ダイアログ ボックスで [Property display level] → [Advanced] をクリックします。
3. [Hierarchy Separator] プロパティを設定します。

アーキテクチャ サポート

アーキテクチャに依存しません。

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

-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。

構文例

xst run -hierarchy_separator _

階層区切り文字を _ (アンダースコア) に設定しています。

IOSTANDARD (I/O 規格)

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。詳細は、『[制約ガイド](#)』の「IOSTANDARD」を参照してください。

KEEP (キープ)

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。

インプリメンテーション フローでサポートされる値には、true と false のほかに、soft もあります。この値が指定されると、XST は true の場合と同じように該当するネットを保持しますが、最終のネットリストではこのネットに KEEP 制約は付けません。

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。

詳細は、『[制約ガイド](#)』の「KEEP」を参照してください。

KEEP_HIERARCHY (階層の維持)

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。

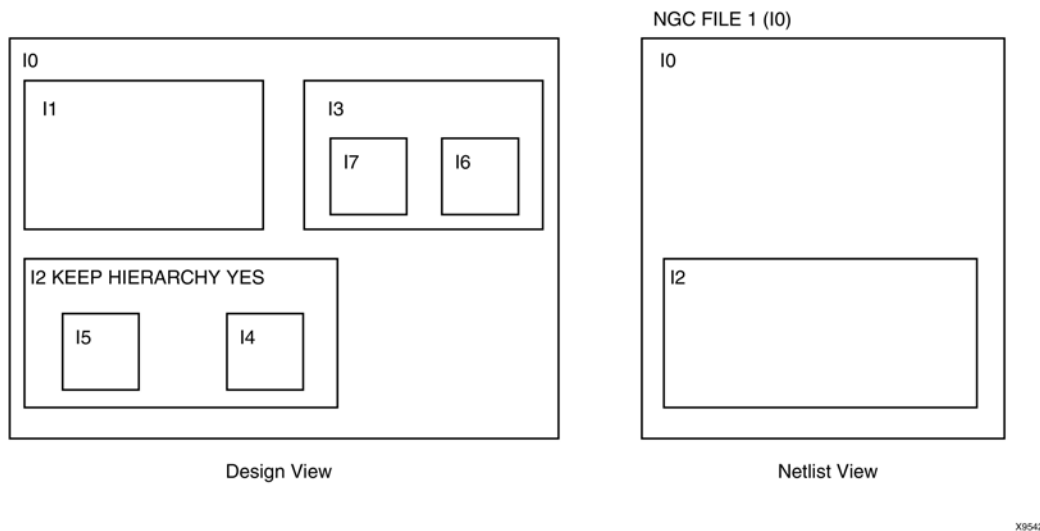
この制約には、次の 3 つの値を使用できます。

- ・ **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。
CPLD デバイスの場合、デフォルトは true です。
- ・ **false**
階層ブロックが 最上位モジュールにマージされます。
FPGA デバイスの場合、デフォルトは false です。
- ・ **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。

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)。

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。

KEEP_HIERARCHY の図



アーキテクチャ サポート

アーキテクチャに依存しません。

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。

回路図からの設定

- ・ エンティティまたはモジュール シンボルに設定
- ・ 属性名 : KEEP_HIERARCHY
- ・ 属性値 : YES、NO

VHDL 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}";
```

FPGA デバイスの場合、デフォルトは no です。

CPLD デバイスの場合、デフォルトは yes です。

Verilog 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。

```
(* keep_hierarchy = "{yes|no|true|false|soft}" *)
```

XST Constraint File (XCF) 構文例

```
MODEL "entity_name" keep_hierarchy={yes|no|true|false|soft};
```

XST コマンドライン

XST コマンドラインで次のように定義します。

```
-keep_hierarchy {yes|no|soft}
```

FPGA デバイスの場合、デフォルトは no です。

CPLD デバイスの場合、デフォルトは yes です。

詳細は、「[コマンドライン モード](#)」を参照してください。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

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

-lso (ライブラリの検索順)

-lso を使用すると、ライブラリ ファイルを使用する順序を指定できます。

LSO ファイルの詳細は、「[混合言語プロジェク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 からライブラリ検索順 (LSO) ファイルを指定するには

1. [Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスを表示して、[Synthesis Options] をクリックします。
2. [Process Properties] ダイアログ ボックスで [Property display level] → [Advanced] をクリックします。
3. [Library Search Order] プロパティを設定します。

アーキテクチャ サポート

アーキテクチャに依存しません。

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

```
-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。

構文例

```
xst elaborate -lso c:/data/my_libraries/my.lso
```

ライブラリ検索順序を設定するファイルに `c:/data/my_libraries/my.lso` を指定しています。

LOC

LOC 制約は、FPGA/CPLD 内でデザイン 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。

-netlist_hierarchy (ネットリスト階層)

-netlist_hierarchy を使用すると、最終の NGC ネットリスト ファイルが生成される形式を制御できます。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、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。

ISE® Design Suite からこのオプションを設定するには

1. [Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスを表示して、[Synthesis Options] をクリックします。
2. [Process Properties] ダイアログ ボックスで [Property display level] → [Advanced] をクリックします。
3. [Netlist Hierarchy] プロパティを設定します。

構文

```
-netlist_hierarchy {as_optimized|rebuilt}
```

デフォルトは as_optimized です。

構文例

```
-netlist_hierarchy rebuilt
```

XST では、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。

OPT_LEVEL (最適化エフォート レベル)

OPT_LEVEL を使用すると、合成最適化のエフォートレベルを指定できます。

OPT_LEVEL に使用できる値は、次のとおりです。

- ・ **1** (normal optimization)

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** (higher optimization)

時間はかかりますが、スライス/マクロセルの数や最大周波数で良い結果が得られます。2 を選択すると、合成のランタイムが長引きます。また、常に最適の結果が得られるとは限りません。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体、エンティティ、モジュールに適用されます。

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、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 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 構文例

次をモジュール宣言またはインスタンスーションの直前に入力します。

```
(* opt_level = "{1 | 2}" *)
```

XST Constraint File (XCF) 構文例

```
MODEL "entity_name" opt_level={1 | 2};
```

XST コマンドライン

この制約は、-opt_level コマンドライン オプションでグローバルに設定できます。

```
-opt_level {1 | 2}
```

デフォルトは 1 です。

ISE Design Suite 構文例

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Optimization Effort]

OPT_MODE (最適化方法の指定)

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

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体、エンティティ、モジュールに適用されます。

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 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 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。

```
(* opt_mode = "{speed | area}" *)
```

XST Constraint File (XCF) 構文例

```
MODEL "entity_name" opt_mode={speed | area};
```

XST コマンドライン

XST コマンドラインで次のように定義します。

```
xst run-opt_mode {area | speed}
```

デフォルトは、speed です。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Optimization Goal]

PARALLEL_CASE (パラレル ケース)

PARALLEL_CASE は、Verilog デザインにのみ使用できます。PARALLEL_CASE を使用すると、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できます。詳細は、「[マルチプレクサの Hardware Description Language \(HDL\) 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

アーキテクチャに依存しません。

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 構文例

構文は次のとおりです。

```
(* parallel_case *)
```

PARALLEL_CASE 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 *)
case select
4'b1xxx: res = data1;
4'b01xx: res = data2;
4'b001x: res = data3;
4'b0001: res = data4;
endcase
```

PARALLEL_CASE 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 parallel_case
```

PARALLEL_CASE 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。

```
case select // synthesis parallel_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

XST コマンド ラインで次のように定義します。

```
xst run -vlgcase {full | parallel | full-parallel}
```

RLOC (RLOC)

RLOC 制約は、基本的なマップおよび配置の制約です。この制約は、ロジック エレメントを独立した集合にグループ化します。デザイン内での最終的な配置に関係なく、集合内でのエレメント同士の位置を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。

S (保存)

S (Save) 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たり、LUT のようなエレメントが最適化で削除されることがあります。このようにブロックに含まれたネットや削除されたネットは、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。S (SAVE) を使用すると、これが回避できます。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にも S (SAVE) 制約によってオフにできるものがあります。

S (SAVE) 制約がネットに適用されると、そのネットは直接接続されたエレメントすべてと共に最終ネットリストに保存されます。これらのエレメントに接続されたネットも保存されます。

S (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。詳細については、『[制約ガイド](#)』を参照してください。

-uc (合成制約ファイルの指定)

-uc を使用すると、合成で使用する XST 合成制約ファイルを指定できます。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。詳細は、『[デザイン制約](#)』を参照してください。

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 [Synthesis Constraint File] でも設定できます。

アーキテクチャ サポート

アーキテクチャに依存しません。

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

The command line syntax is:

```
xst run -uc filename
```

構文例

```
-uc my_constraints.xcf
```

このプロジェクトの制約ファイルに my_constraints.xcf を指定しています。

TRANSLATE_OFF (変換なし) および TRANSLATE_ON (変換あり)

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 VHDL または Verilog 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 により 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

アーキテクチャに依存しません。

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 構文例

VHDL では、TRANSLATE_OFF を次のように記述します。

```
-- synthesis translate_off
...code not synthesized...
-- synthesis translate_on
```

Verilog 構文例

TRANSLATE_OFF は、VHDL または Verilog のメタ コメントとして使用できます。ただし、Verilog 構文は、前述したような通常のメタ 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
```

-iuc (合成制約ファイルの使用)

-iuc を使用すると、合成中に制約ファイルが無視できます。

アーキテクチャ サポート

アーキテクチャに依存しません。

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。

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 [Use Synthesis Constraints File] で設定できます。

構文

```
xst run -iuc {yes / no}
```

デフォルトでは no に設定されています。

構文例

```
xst run -iuc yes
```

合成中に制約ファイルが使用されます。

-vlgincdir (Verilog の 'include ディレクトリの指定)

-vlgincdir を使用すると、Verilog の 'include ディレクトリのパスを指定し、パーサーにファイルを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

アーキテクチャに依存しません。

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。

ISE® Design Suite で Verilog の 'include ディレクトリを指定するには

1. [Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスを表示して、[Synthesis Options] をクリックします。
2. [Process Properties] ダイアログ ボックスで [Property display level] → [Advanced] をクリックします。
3. [Verilog Include Directories] プロパティを設定します。

構文

```
-vlgincdir {directory_path [directory_path]}
```

directory_path はディレクトリ名です。詳細は、「[コマンドライン モードでのスペースを含む名前](#)」を参照してください。

構文例

```
xst elaborate -vlgincdir c:/my_verilog
```

XST がファイルを検索するディレクトリのリストに c:/my_verilog を追加します。

-verilog2001 (Verilog 2001 の指定)

-verilog2001 を使用すると、Verilog ソース コードを Verilog 2001 規格として解釈するかどうかを指定できます。デフォルトでは、Verilog ソース コードは Verilog 2001 規格として解釈されます。

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 [Verilog 2001] でも設定できます。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

ありません。

適用ルール

ありません。

構文

```
xst run -verilog2001 {yes|no}
```

デフォルトでは、yes に設定されています。

構文例

```
xst elaborate -verilog2001 no
```

XST では、Verilog コードが Verilog 2001 規格に従って解釈されません。

-xsthdpini (HDL ライブラリ マップ ファイル)

-xsthdpini を使用すると、ライブラリのマップに使用するファイルを選択できます。

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。このファイルは修正するべきではありませんが、ユーザーのライブラリ マップ ファイルの構文をコピーすることはできます。

ISE® Design Suite からライブラリ マップ ファイルを設定するには

1. ISE Design Suite の [Process Properties] ダイアログ ボックスを表示して、[Synthesis Options] をクリックします。
2. [Process Properties] ダイアログ ボックスで [Property display level] → [Advanced] をクリックします。
3. [HDL INI File] プロパティを設定します。

ライブラリ マップ ファイルは次のようになっています。

```
-- 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 ISE Design Suite 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。

このライブラリ マップ ファイルには、次の情報別にライブラリがリストされています。

- ・ 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

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 -- を使用します。

アーキテクチャ サポート

アーキテクチャに依存しません。

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

-xsthdpini *file_name*

ライブラリ マップ ファイルは 1 つだけしか指定できません。

構文例

```
xst set -xsthdpini c:/data/my_libraries/my.ini file_name
```

ライブラリすべてをポイントするファイルに c:/data/my_libraries/my.ini を指定しています。

run コマンドを実行する前に、この set コマンドを実行する必要があります。

MY.INI ファイル例

```
work1=H:\Users\conf\my_lib\work1  
work2=C:\mylib\work2
```

-xsthdpdir (作業ディレクトリ)

-xsthdpdir (作業ディレクトリ) では、ライブラリ マップ ファイルで場所が指定されていない場合に、コンパイルされたファイルを保存する場所を指定します。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 から [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] にある [VHDL Working Directory] で設定
- 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
```

-xsthdpdir (作業ディレクトリ) の例

この例は、次の条件下にあります。

- 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

アーキテクチャに依存しません。

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

XST コマンド ラインの構文例

run コマンドを実行する前に、**set -xsthdpdir** を使用して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 で次のように定義します。

[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] → [VHDL Work Directory] で設定します。

このプロパティは、次の設定で表示されます。

[Process Properties] ダイアログ ボックス → [Property display level] → [Advanced]

Hardware Description Language (HDL) 制約

このセクションでは、XST で使用できる Hardware Description Language (HDL) デザイン制約について説明します。

- ・ [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 \(MUX_EXTRACT\)](#)
- ・ [レジスタ電源投入時の値 \(REGISTER_POWERUP\)](#)
- ・ [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 \(SAFE_IMPLEMENTATION\)](#)
- ・ [信号のエンコード方法 \(SIGNAL_ENCODING\)](#)

この章で説明される制約は、次に適用されます。

- ・ FPGA デバイス
- ・ CPLD デバイス
- ・ VHDL コード
- ・ Verilog コード

ここに示すほとんどの制約は、ISE® Design Suite の [Process Properties] ダイアログ ボックスの [HDL Options] ページでグローバルに設定できます。この方法で設定できない制約は、次の 3 つです。

- ・ [列挙型エンコード手法 \(ENUM_ENCODING\)](#)
- ・ [セーフ リカバリ ステート \(SAFE_RECOVERY_STATE\)](#)
- ・ [信号のエンコード方法 \(SIGNAL_ENCODING\)](#)

FSM_EXTRACT (FSM 自動抽出)

FSM_EXTRACT を使用すると、有限状態 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、この制約をオンにする必要があります。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、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、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 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 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}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" fsm_extract={yes | no | true | false};
```

XST Constraint File (XCF) 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

XST コマンドラインで次のように定義します。

```
-fsm_extract {yes | no}
```

デフォルトでは、yes に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックスの [HDL Options] → [FSM Encoding Algorithm] から設定できます。選択する値により、次のように設定されます。

- ・ [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。
- ・ ほかの値を選択すると、FSM_EXTRACT は yes に設定され、FSM_ENCODING には選択された値が設定されます。**-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。

ENUM_ENCODING (列挙型エンコード手法)

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。ENUM_ENCODING は、列挙タイプで VHDL 制約としてのみ指定できます。

ステートレジスタの列挙タイプを使用して 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

アーキテクチャに依存しません。

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

設定し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 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
...
```

XST Constraint File (XCF) 構文例

```
BEGIN MODEL "entity_name" NET "signal_name" enum_encoding="string";END;
```

EQUIVALENT_REGISTER_REMOVAL (等価レジスタの削除)

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。フリップフロップの最適化では、次が削除されます。

- ・ FPGA および CPLD の等価フリップフロップ
- ・ CPLD の定数入力を使用するフリップフロップ

フリップフロップを削除することでロジックが単純化され、フィットが向上します。

使用できる値は、次のいずれかです。

- ・ **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

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

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 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 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}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL " entity_name " equivalent_register_removal= {yes | no | true | false};
```

XST Constraint File (XCF) 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

XST コマンド ラインで次のように定義します。

```
-equivalent_register_removal {yes | no}
```

デフォルトでは、yes に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Equivalent Register Removal]

FSM_ENCODING (FSM エンコード方法の指定)

FSM_ENCODING を使用すると、Finite State Machine (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 yes にしておく必要があります。

設定可能な値は、次のいずれかです。

- ・ Auto
- ・ One-Hot
- ・ Compact
- ・ Sequential
- ・ Gray
- ・ Johnson
- ・ Speed1
- ・ User

デフォルトは Auto で、ステート マシンごとに適したエンコード方法が自動的に指定されます。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、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、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 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}";
```

デフォルトでは、auto に設定されています。

Verilog 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。

次をモジュールまたは信号宣言の直前に入力します。

```
(* fsm_encoding = "{auto | one-hot | compact | sequential | gray | johnson | speed1 | user}" *)
```

デフォルトでは、auto に設定されています。

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" fsm_encoding={auto | one-hot | compact | sequential |  
gray | johnson | speed1 | user} ;
```

XST Constraint File (XCF) 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

XST コマンドラインで次のように定義します。

```
-fsm_encoding {auto | one-hot | compact | sequential | gray | johnson | speed1 | user}
```

デフォルトでは、auto に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックスの [HDL Options] → [FSM Encoding Algorithm] から設定できます。

選択する値により、次のように設定されます。

- ・ [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。
- ・ ほかの値を選択すると、`FSM_EXTRACT` は `yes` に設定され、`FSM_ENCODING` には選択された値が設定されます。詳細は、「[FSM 自動抽出 \(FSM_EXTRACT\)](#)」を参照してください。

MUX_EXTRACT (MUX の抽出)

`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。

`MUX_EXTRACT` の値には、次を設定できます。

- ・ **yes**
- ・ **no**
- ・ **force**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

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、または残りのロジックと共に最適化されます。`force` に設定すると、この内部ルールが無視され、マクロが作成されます。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、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、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 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}";
```

デフォルトでは、`yes` に設定されています。

Verilog 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}" *)
```

デフォルトでは、`yes` に設定されています。

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" mux_extract={yes | no | true | false | force};
```

XST Constraint File (XCF) 構文例 2

```
BEGIN MODEL "entity_name" NET "signal_name" mux_extract={yes | no | true | false | force};END;
```

XST コマンド ライン

XST コマンド ラインで次のように定義します。

```
-mux_extract {yes | no | force}
```

デフォルトでは、yes に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [Mux Extraction]

REGISTER_POWERUP (レジスタ電源投入時の値)

XSTでは、レジスタの電源投入時の値は自動的に追加されません。このため、REGISTER_POWERUP 制約を使用し、値を明示的に指定する必要があります。この制約は、VHDL の列挙型に割り当てるか、VHDL 属性を使用して VHDL の信号に直接設定するか、または Verilog メタ コメントを使用して Verilog のレジスタ ノードに直接設定します。制約の値は、2 進数の文字列またはシンボル コード値で指定します。

アーキテクチャ サポート

次のデバイスにのみ適用できます。これ以外のデバイスには適用できません。

- ・ CPLD デバイスすべて
- ・ Spartan®-3A デバイス

適用可能エレメント

信号または型に適用されます。

適用ルール

設定し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 構文例 1

std_logic_vector など、定義済みの VHDL 型でレジスタを定義します。REGISTER_POWERUP の値はバイナリコードにする必要があります。

```
signal myreg : std_logic_vector (3 downto 0); attribute register_powerup of myreg : signal is "0001";
```


VHDL 構文例 2

レジスタを列挙型 (シンボリック ステート マシン) で定義します。REGISTER_POWERUP 制約は信号に設定し、その値は定義されたシンボリック ステートの 1 つに指定します。実際の電源投入コードは、ステート マシンのエンコード方法によって異なります。

```
type state_type is (s1, s2, s3, s4, s5); signal statel : state_type;
```

VHDL 構文例 3

REGISTER_POWERUP 制約は列挙型に設定され、その型に定義されたレジスタすべてにこの制約が使用されます。

```
type state_type is (s1, s2, s3, s4, s5);
attribute register_powerup of state_type : type is "s1";
signal statel, state2 : state_type;
```

VHDL 構文例 4

列挙型のオブジェクトの場合、電源投入時の値は 2 進数コードとしても定義できます。ただし、自動エンコードを使用しており、異なるエンコード方法 (特に異なるコード幅) が適用される場合は、この電源投入時の値は無視されます。

```
type state_type is (s1, s2, s3, s4, s5);
attribute enum_encoding of state_type : type is "001 011 010 100 111";
attribute register_powerup of state_type : type is "100";
signal statel : state_type;
```

Verilog 構文例

REGISTER_POWERUP を信号の直前に記述します。

```
(* register_powerup = "<value>" *)
```

XST Constraint File (XCF) 構文例

```
BEGIN MODEL "entity_name "
NET "signal_name" register_powerup="string";
END;
```

RESOURCE_SHARING (リソース共有)

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。

RESOURCE_SHARING には、次の値を使用できます。

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **force**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体またはデザイン エレメントに適用されます。

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、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 構文例

次のように宣言します。

```
attribute resource_sharing: string;
```

次のように指定します。

```
attribute resource_sharing of entity_name: entity is "{yes | no}";
```

Verilog 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。

```
(* resource_sharing = "{yes | no}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" resource_sharing={yes | no | true | false};
```

XST Constraint File (XCF) 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

XST コマンドラインで次のように定義します。

```
-resource_sharing {yes | no}
```

デフォルトでは、yes に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [Resource Sharing]

SAFE_RECOVERY_STATE (セーフ リカバリ ステート)

SAFE_RECOVERY_STATE を使用すると、有限ステート マシン (Finite State Machine (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

アーキテクチャに依存しません。

適用可能エレメント

ステートレジスタを表す信号に適用されます。

適用ルール

設定された信号に適用されます。

構文例

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 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 構文例

次を信号宣言の直前に入力します。

```
(* safe_recovery_state = "<value>" *)
```

XST Constraint File (XCF) 構文例

```
BEGIN MODEL "entity_name "
```

```
NET "signal_name" safe_recovery_state="<value>";
```

```
END;
```

SAFE_IMPLEMENTATION (セーフ インプリメンテーション)

SAFE_IMPLEMENTATION を使用すると、有限ステートマシン (Finite State Machine (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。

SAFE_IMPLEMENTATION を使用するには、次の手順に従ってください。

- ISE® Design Suite
[Process Properties] ダイアログ ボックス → [HDL Options] → [Safe Implementation] をクリックします。
- Hardware Description Language (HDL)
ステートレジスタを表す階層ブロックまたは信号に SAFE_IMPLEMENTATION 制約を設定します。

アーキテクチャ サポート

アーキテクチャに依存しません。

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、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 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 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}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" safe_implementation={yes | no | true | false};
```

XST Constraint File (XCF) 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

XST コマンド ラインで次のように定義します。

```
-safe_implementation {yes | no}
```

デフォルトでは no に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [Safe Implementation]

SIGNAL_ENCODING (信号のエンコード方法)

SIGNAL_ENCODING では、内部信号に使用するコーディング手法を指定します。

SIGNAL_ENCODING には、次の値を使用できます。

- ・ **auto**
デフォルトです。各信号に対して最適なコーディング手法が自動的に選択されます。
- ・ **one-hot**
ワンホット エンコード方法が使用されます。
- ・ **user**
ユーザーのエンコード方法が使用されます。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、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、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 構文例

次のように宣言します。

```
attribute signal_encoding: string;
```

次のように指定します。

```
attribute signal_encoding of {component_name | signal_name | entity_name | label_name} :  
{component | signal | entity | label} is "{auto | one-hot | user}";
```

デフォルトでは、auto に設定されています。

Verilog 構文例

次を信号宣言の直前に入力します。

```
(* signal_encoding = "{auto | one-hot | user}" *)
```

デフォルトでは、auto に設定されています。

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" signal_encoding = {auto | one-hot | user};
```

XST Constraint File (XCF) 構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" signal_encoding = {auto | one-hot | user};  
END;
```

XST コマンド ライン

XST コマンド ラインで次のように定義します。

```
-signal_encoding {auto | one-hot | user}
```

デフォルトでは、auto に設定されています。

FPGA 制約 (タイミング制約以外)

多くの場合、制約はエンティティまたはモデル全体にグローバルに適用するか、または個々の信号、ネット、またはインスタンスにローカルに適用できます。有効な制約ターゲットについては、「[XST 固有のオプション \(タイミング以外\)](#)」および「[XST 固有のオプション \(タイミング以外\) : XST コマンド ラインのみ](#)」を参照してください。

次の XST の FPGA 制約 (タイミング以外) は、FPGA デバイスのみに適用されます。これらの制約は、CPLD には適用できません。

- ・ 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)
- ・ デコーダの抽出 (DECODER_EXTRACT)
- ・ 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)
- ・ 論理シフタの抽出 (SHIFT_EXTRACT)
- ・ 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)
- ・ MUX スタイル (MUX_STYLE)
- ・ グローバル クロック バッファ数 (-bufg)
- ・ リージョン クロック バッファ数 (-bufr)
- ・ インスタンシエート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)
- ・ プライオリティ エンコーダの抽出 (PRIORITY_EXTRACT)
- ・ 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)
- ・ XOR コラプス (XOR_COLLAPSE)
- ・ スライス (LUT-FF ペア) 使用率 (SLICE_UTILIZATION_RATIO)
- ・ スライス (LUT-FF ペア) 使用率の許容範囲 (SLICE_UTILIZATION_RATIO_MAXMARGIN)
- ・ 単一 LUT へのエンティティのマップ (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)
- ・ DSP48 の使用 (USE_DSP48)

ASYNC_TO_SYNC (非同期から同期への変換)

ASYNC_TO_SYNC を使用すると、デザイン全体の非同期セット/リセット信号を同期信号に置き換えることができます。これにより、DSP48 および BRAM にレジスタを組み込んで結果を改善可能です。電力最適化にも好影響を与えることができます。

XST では基本的に BRAM に FSM を配置できますが、ほとんどの場合 FSM には非同期セット/リセットが使用されており、こういった FSM は BRAM にはインプリメントできません。ASYNC_TO_SYNC 制約を使用すると、BRAM に FSM を簡単に配置できるので、手動でデザインを変更する必要がなくなります。

非同期セット/リセット信号を同期信号に置換すると、生成した NGC ネットリストが最初の RTL 記述と同じではなくなります。合成したデザインが最初の仕様を満たしているかどうか必ず確認してください。異なる場合、XST では次のような警告メッセージが表示されます。

WARNING: You have requested that asynchronous control signals of sequential elements be treated as if they were synchronous. If you haven't done so yet, please carefully review the related documentation material. If you have opted to asynchronously control flip-flop initialization, this feature allows you to better explore the possibilities offered by the Xilinx solution without having to go through a painful rewriting effort. However, be well aware that the synthesis result, while providing you with a good way to assess final device usage and design performance, is not functionally equivalent to your HDL description. As a result, you will not be able to validate your design by comparison of pre-synthesis and post-synthesis simulation results. Please also note that in general we strongly recommend synchronous flip-flop initialization.

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

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

XST コマンドラインで次のように定義します。

```
xst run -async_to_sync{yes | no}
```

デフォルトでは no に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [Asynchronous to Synchronous]

AUTO_BRAM_PACKING (自動 BRAM パッキング)

2 つの小型 BRAM をデュアル ポート BRAM として 1 つの BRAM プリミティブにパッキングできます。XST では BRAM 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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

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

XST コマンドラインで次のように定義します。

```
-auto_bram_packing {yes | no}
```

デフォルトでは no に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [Automatic BRAM Packing]

BRAM_UTILIZATION_RATIO (BRAM 使用率)

BRAM_UTILIZATION_RATIO を使用すると、合成中に XST で処理される BRAM ブロック数を制限できます。デザインに含まれる BRAM は BRAM の推論からだけでなく、インスタンス化と BRAM 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 BRAM にマップされるようにします。詳細は、「[ブロック RAM へのロジックのマップ](#)」を参照してください。

インスタンス化された BRAM は使用可能な BRAM リソースの第一候補として認識され、BRAM が推論されると、残りの BRAM リソースに配置されます。インスタンス化された BRAM 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 BRAM 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、BRAM 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 BRAM の数がターゲット FPGA デバイスの BRAM 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、使用可能な BRAM リソースのみが自動的に使用されます。自動的にリソースが管理されないようにするには、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 BRAM 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 BRAM 数がターゲット FPGA で使用可能な BRAM 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 BRAM がすべて分散 RAM に変換され、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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

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

XST コマンドラインで次のように定義します。

```
-bram_utilization_ratio <integer>[%][#]
```

この場合、それぞれ次を表します。

<integer> の範囲は、% が使用されるか、% と # のどちらも削除される場合は -1 ~ 100 になります。

デフォルトは 100 です。

XST コマンド ライン例 1

```
-bram_utilization_ratio 50
```

ターゲット デバイスで BRAM ブロックの 50% が使用されます。

XST コマンド ライン例 2

```
-bram_utilization_ratio 50%
```

ターゲット デバイスで BRAM ブロックの 50% が使用されます。

XST コマンド ライン例 3

```
-bram_utilization_ratio 50#
```

50 個の BRAM ブロックが使用されます。

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。

XST で推論される BRAM の数を確認する場合などは、この BRAM 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。

BUFFER_TYPE (バッファ タイプ)

BUFFER_TYPE 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 制約は、入力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。値 bufr は、Virtex®-4 および Virtex-5 デバイス ファミリでのみ使用できます。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

信号に適用されます。

適用ルール

設定された信号に適用されます。

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 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 " {bufgdll | ibufg | bufgp | ibuf | bufr | none}";
```

Verilog 構文例

次を信号宣言の直前に入力します。

```
(* buffer_type = "{bufgdll | ibufg | bufgp | ibuf | bufr | none}" *)
```

XST Constraint File (XCF) 構文例

```
BEGIN MODEL "entity_name" NET "signal_name" buffer_type={bufgdll | ibufg  
| bufgp | ibuf | bufr | none}; END;
```

BUFGCE (BUFGCE の抽出)

BUFGCE を使用すると、BUFGMUX プリミティブを推論し、BUFGMUX の機能をインプリメントできます。この動作によって、ワイヤ数が低減されます。クロック信号およびクロック イネーブル信号は、1 本のワイヤで N 個の順序コンポーネントに送られます。

この制約は、プライマリ クロック信号に設定する必要があります。

使用できる値は、次のいずれかです。

- ・ **yes**
- ・ **no**

BUFGCE、Hardware Description Language (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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

クロック信号に適用されます。

適用ルール

設定された信号に適用されます。

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 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 構文例

次を信号宣言の直前に入力します。

```
(* bufgce = "{yes | no}" *)
```

XST Constraint File (XCF) 構文例

```
BEGIN MODEL "entity_name" NET "primary_clock_signal" bufgce={yes | no | true | false};END;
```

-sd (コアの検索ディレクトリ)

-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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

ISE® Design Suite の [Process Properties] ダイアログ ボックスの [Synthesis Options] ページにある [Cores Search Directory] で設定します。

構文

```
-sd {directory_path [directory_path]}
```

There is no default.

構文例

```
xst run -sd c:/data/cores c:/ise/cores
```

コアがデフォルトのディレクトリだけでなく、c:/data/cores および c:/ise/cores でも検索されます。詳細は、「[コマンドライン モードでのスペースを含む名前](#)」を参照してください。

デコーダの抽出 (DECODER_EXTRACT)

デコーダ マクロの推論を有効または無効にします。

デコーダの抽出 (DECODER_EXTRACT) のアーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

デコーダの抽出 (DECODER_EXTRACT) の適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

デコーダの抽出 (DECODER_EXTRACT) の伝播規則

ネットまたは信号に設定すると、そのネットまたは信号に適用されます。

エンティティまたはモジュールに設定すると、そのエンティティまたはモジュールの階層内にあるすべての適用可能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 構文例

次のように宣言します。

```
attribute decoder_extract: string;
```

次のように指定します。

```
attribute decoder_extract of {entity_name | signal_name}: {entity | signal} is "{yes | no}";
```

Verilog 構文例

次をモジュール宣言または信号宣言の直前に入力します。

```
(* decoder_extract "{yes | no}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" decoder_extract={yes | no | true | false};
```

XST Constraint File (XCF) 構文例 2

```
BEGIN
```

```
MODEL "entity_name" NET "signal_name" decoder_extract={yes | no | true | false};
```

```
END;
```

XST コマンド ライン

XST コマンド ラインで次のように定義します。

```
xst run -decoder_extract {yes | no}
```

デフォルトでは、yes に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] → [HDL Options] → [Decoder Extraction] で設定します。

使用できる値は次のいずれかです。

- ・ **yes** (デフォルト)
- ・ **no** (チェックボックスはオフ)

DSP_UTILIZATION_RATIO (DSP 使用率)

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。[DSP48 の使用 \(USE_DSP48\)](#) 制約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 FPGA デバイスの 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 リソースのみが使用されて合成されます。

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

次の FPGA デバイスでのみ使用できます。これ以外の FPGA には適用できません。CPLD には適用できません。

- ・ Spartan®-3A DSP
- ・ Virtex®-4
- ・ Virtex-5

適用可能エレメント

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

XST コマンドラインで次のように定義します。

```
-dsp_utilization_ratio integer[%|#]
```

この場合、それぞれ次を表します。

<integer> 範囲は、% が使用されるか、% と # のどちらも削除される場合は -1 ~ 100 になります。

合計スライスの割合を指定するには % 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。

たとえば、次のように指定します。

- ・ 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#
```

メモ： 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_STYLE (FSM スタイル)

FSM_STYLE を使用すると、Virtex® デバイスおよびそれ以降のデバイスに搭載されているブロック RAM リソースを使用してインプリメントすることで、大型の Finite State Machine (FSM) コンポーネントをさらにコンパクトに高速にできます。ブロック RAM リソースを使用すると、エリアを小さくでき、また速度も向上します。デフォルトでは、FSM は LUT を使用してインプリメントされます。グローバルに、またはローカルに設定できます。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、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、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 構文例

次のように宣言します。

```
attribute fsm_style: string;
```

次のように宣言します。

```
attribute fsm_style of {entity_name | signal_name}: {entity | signal} is "{lut | bram}";
```

デフォルトは lut です。

Verilog 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_style = "{lut | bram}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" fsm_style = {lut | bram};
```

XST Constraint File (XCF) 構文例 2

```
BEGIN MODEL "entity_name" NET "signal_name" fsm_style = {luxt | bram};END;
```

XST Constraint File (XCF) 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 で次のように定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [FSM Style]

POWER (電力削減)

POWER 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。この制約は、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 BRAM 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 BRAM 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 BRAM 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®-4 および Virtex-5 デバイスにのみ適用できます。これ以外の FPGA には適用できません。CPLD には適用できません。

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、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 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}";
```

デフォルトでは no に設定されています。

Verilog 構文例

この制約は、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}" *)
```

デフォルトでは no に設定されています。

XST Constraint File (XCF) 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

XST コマンド ラインで次のように定義します。

```
xst run -power {yes | no}
```

デフォルトでは no に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Power Reduction] をクリックします。

READ_CORES (コアの読み込み)

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。特定のコアが XST に読み込んだ方がロジックの接続がわかるので、そのコアの周囲のロジックを最適化しやすくなりますが、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。READ_CORES 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。

詳細については、「[コアの処理](#)」を参照してください。

READ_CORES の値には、次のいずれかを指定できます。

- ・ **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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

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。

READ_CORES は次に適用できます。

- ・ 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 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} : {component | entity}  
is "{yes | no | optimize}";
```

デフォルトでは、yes に設定されています。

Verilog 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}" *)
```

デフォルトでは、yes に設定されています。

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" read_cores = {yes | no | true | false | optimize};
```

XST Constraint File (XCF) 構文例 2

```
BEGIN MODEL "entity_name" END;
```

```
INST "instance_name" read_cores = {yes | no | true | false | optimize};
```

```
END;
```

デフォルトでは、yes に設定されています。

XST コマンドライン

XST コマンドラインで次のように定義します。

```
-read_cores {yes | no | optimize}
```

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Read Cores] をクリックします。

ISE Design Suite からは、optimize オプションは指定できません。

SHIFT_EXTRACT (論理シフタの抽出)

SHIFT_EXTRACT を使用すると、論理シフタ マクロの推論を有効または無効にできます。

SHIFT_EXTRACT の値は次のいずれかになります。

- ・ **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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、またはデザイン エレメント、ネットに適用されます。

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、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 構文例

次のように宣言します。

```
attribute shift_extract: string;
```

次のように指定します。

```
attribute shift_extract of {entity_name | signal_name}: {signal | entity} is "{yes | no}";
```

Verilog 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。

```
(* shift_extract = "{yes | no}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" shift_extract={yes | no | true | false};
```

XST Constraint File (XCF) 構文例 2

```
BEGIN MODEL "entity_name "
```

```
NET "signal_name" shift_extract={yes | no | true | false};
```

```
END;
```

XST コマンド ライン

XST コマンド ラインで次のように定義します。

```
-shift_extract {yes | no}
```

デフォルトでは、yes に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [Logical Shifter Extraction]

LC (LUT の結合)

共通の入力を持つ LC (LUT Combining) 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。
- ・ **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**
LC はディスエーブルにされます。

アーキテクチャ サポート

Virtex®-5 デバイスにのみ適用できます。これ以外の FPGA には適用できません。CPLD には適用できません。

適用可能エレメント

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

XST コマンド ラインで次のように定義します。

```
xst run -lc {auto | area | off}
```

デフォルトは off です。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [LUT Combining]

BRAM_MAP (BRAM へのロジックのマッピング)

BRAM_MAP を使用すると、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 リソースにマップするよう指定できます。

BRAM_MAP の値は、次のとおりです。

- ・ **yes**
- ・ **no** (デフォルト)

BRAM_MAP は、グローバルに、またはローカルに設定できます。詳細は、「[ブロック RAM へのロジックのマッピング](#)」を参照してください。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

BRAM に適用されます。

適用ルール

BRAM にマップするロジック (出力レジスタを含む) は、異なる階層レベルに指定する必要があります。ロジックが 1 つの BRAM にフィットしないと、BRAM にマップされません。エンティティ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。BRAM が推論されない場合、ロジックがグローバルに最適化され、マクロは推論されません。XST によりロジックがマップされていることを確認してください。

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 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 構文例

次をモジュール宣言またはインスタンス化の直前に入力します。

```
(* bram_map = "{yes | no}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" bram_map = {yes | no | true | false};
```

XST Constraint File (XCF) 構文例 2

```
BEGIN MODEL "entity_name"
```

```
  INST "instance_name" bram_map = {yes | no | true | false};
```

```
END;
```

MAX_FANOUT (最大ファンアウト数)

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。デフォルトは、次の表に示すようにターゲット デバイス ファミリによって異なります。MAX_FANOUT はグローバルにも、またはローカルにも設定できます。

デフォルト値

デバイス	デフォルト値
Spartan®-3	500
Spartan-3E	
Spartan-3A	
Spartan-3A DSP	
Virtex®-4	500
Virtex-5	100000 (10 万)

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 30 未満に設定されている場合などは、厳密に適用されるとは限りません。

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。これらのバッファには、NGC ファイルで **キープ (KEEP)** 属性が設定され、インプリメンテーションでの最適化により削除されることはありません。

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、指定した信号ごとに設定して、最大ファンアウトを制御できます。

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。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体に適用されます。

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、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 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";
```

Verilog 構文例

次を信号宣言の直前に入力します。

```
(* max_fanout = "integer" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" max_fanout=integer;
```

XST Constraint File (XCF) 構文例 2

```
BEGIN MODEL "entity_name"
```

```
  NET "signal_name" max_fanout=integer;
```

```
END;
```

XST コマンド ライン

XST コマンド ラインで次のように定義します。

```
-max_fanout integer
```

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Max Fanout]

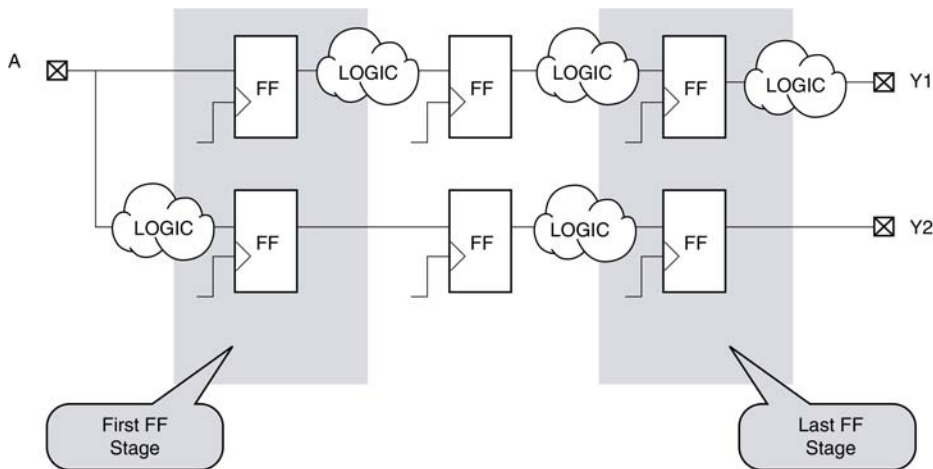
MOVE_FIRST_STAGE (最初のフリップフロップ ステージの移動)

MOVE_FIRST_STAGE を使用すると、プライマリ入力からのレジスタのリタイミングを制御できます。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。

メモ :

- ・ 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。

MOVE_FIRST_STAGE の図



X9564

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。

- ・ デザインに必須の要件がない
- ・ 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

次の 2 つの制約を使用できます。

- ・ MOVE_FIRST_STAGE
- ・ MOVE_LAST_STAGE

どちらの制約も、yes または no に設定できます。

- ・ **MOVE_FIRST_STAGE=no**
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=no**
最後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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

MOVE_FIRST_STAGE は、次にのみ適用できます。

- ・ 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 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 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}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" move_first_stage={yes | no | true | false};
```

XST Constraint File (XCF) 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

XST コマンドラインで次のように定義します。

```
xst run -move_first_stage {yes | no}
```

デフォルトでは、yes に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [Xilinx Specific Options] → [Move First Flip-Flop Stage]

MOVE_LAST_STAGE (最後のフリップフロップ ステージの移動)

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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

次に適用されます。

- ・ 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。

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 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}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" move_last_stage={yes | no | true | false};
```

XST Constraint File (XCF) 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

XST コマンド ラインで次のように定義します。

```
xst run -move_last_stage {yes | no}
```

デフォルトでは、yes に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [Specific Options] → [Move Last Stage]

MULT_STYLE (乗算器スタイル)

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。

MULT_STYLE には、次の値を設定できます。

- ・ **auto**
デフォルトでは、auto に設定されています。
このデフォルト設定を使用すると、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接続する場合に使用します。これは、Virtex®-4、Virtex-5、Spartan®-3A DSP デバイスにのみ適用されます。
- ・ **kcm**
- ・ **csd**
- ・ **lut**
- ・ **pipe_lut**
スライス ベースの乗算器をパイプライン接続する場合に使用します。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、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、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 構文例

次のように宣言します。

```
attribute mult_style: string;
```

次のように指定します。

```
attribute mult_style of {signal_name | entity_name}: {signal | entity} is "{auto  
| block | pipe_block | kcm | csd | lut | pipe_lut}";
```

Verilog 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}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" mult_style={auto | block | pipe_block | kcm | csd | lut | pipe_lut};
```

XST Constraint File (XCF) 構文例 2

```
BEGIN MODEL "entity_name" NET "signal_name" mult_style={auto | block | pipe_block  
| kcm | csd | lut | pipe_lut}  
;END;
```

XST コマンド ライン

XST コマンド ラインで次のように定義します。

```
xst run -mult_style {auto | block | pipe_block | kcm | csd | lut | pipe_lut}
```

-mult_style コマンド ライン オプションは、Virtex-4、Virtex-5 および Spartan-3A DSP ではサポートされていません。これらのデバイスの場合は、**-use_dsp48** を使用してください。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [Multiplier Style]

MUX_STYLE (MUX スタイル)

MUX_STYLE を使用すると、マルチプレクサ マクロのインプリメント方法を指定できます。

MUX_STYLE の値には、次を設定できます。

- ・ **auto** (デフォルト)
- ・ **muxf**
- ・ **muxcy**

auto は DOCTYPE コマンドです。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。

使用可能なインプリメンテーション スタイル

デバイス	リソース
Spartan®-3	MUXF
Spartan-3E	MUXF6
Spartan-3A	MUXCY
Spartan-3A DSP	MUXF7
Virtex®-4	MUXF8
Virtex-5	

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

適用可能エレメント

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 構文例

次のように宣言します。

```
attribute mux_style: string;
```

次のように指定します。

```
attribute mux_style of {signal_name | entity_name}: {signal | entity} is "{auto | muxf | muxcy}";
```

デフォルトでは、auto に設定されています。

Verilog 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_style = "{auto | muxf | muxcy}" *)
```

デフォルトでは、auto に設定されています。

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" mux_style={auto | muxf | muxcy};
```

XST Constraint File (XCF) 構文例 2

```
BEGIN MODEL "entity_name" NET "signal_name" mux_style={auto | muxf | muxcy};END;
```

XST コマンドライン

XST コマンドラインで次のように定義します。

```
xst run -mux_style {auto | muxf | muxcy}
```

auto は DOCTYPE コマンドです。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [Mux Style]

-bufg (グローバル クロック バッファ数)

-bufg を使用すると、XST で作成される BUFG の最大数を整数で指定できます。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 の最大使用可能数と同じになります。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

ISE® Design Suite でグローバル クロック バッファの数を設定するには

1. [Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックス で [Synthesis Options] → [Specific Options] をクリックします。
2. [Process Properties] ダイアログ ボックスで [Property display level] → [Advanced] をクリックします。
3. [Number of Clock Buffers] プロパティを設定します。

構文

値は整数にします。ターゲット デバイスの BUFG の最大数を超える値は設定できません。

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ようになります。

デバイス	デフォルト値
Virtex®-4	32
Virtex-5	
Spartan®-3	8
Spartan-3E	24
Spartan-3A	
Spartan-3A DSP	

構文例

xst run -bufg 8

グローバル クロック バッファの数を 8 に設定しています。

-bufg (リージョン クロック バッファ数)

-bufg を使用すると、XST で作成される BUFG の最大数を整数で指定できます。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 の最大使用可能数と同じになります。

アーキテクチャ サポート

- ・ Virtex®-4 デバイスでのみ使用可能なことがあります。
- ・ Virtex-5 デバイスでは使用できない可能性があります。
- ・ CPLD には適用できません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

ISE® Design Suite でリージョン クロック バッファの数を設定するには

1. [Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックス で [Synthesis Options] → [Specific Options] をクリックします。
2. [Process Properties] ダイアログ ボックスで [Property display level] → [Advanced] をクリックします。
3. [Number of Regional Clock Buffers] プロパティを設定します。

構文

```
xst run -bufr integer
```

値は整数にします。ターゲット デバイスの BUFR の最大数を超える値は設定できません。

構文例

```
xst run -bufr 6
```

リージョン クロック バッファの数を 6 に設定しています。

OPTIMIZE_PRIMITIVES (インスタンス化されたプリミティブの最適化)

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 プリミティブを最適化できます。

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、BRAM、DSP48 などは、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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

階層ブロック、コンポーネント、およびインスタンスに適用されます。

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。

回路図からの設定

- ・ 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 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 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。

```
(* optimize_primitives = "{yes | no}" *)
```

XST Constraint File (XCF) 構文例

```
MODEL "entity_name" optimize_primitives = {yes | no | true | false};
```

ISE Design Suite 構文例

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Optimize Instantiated Primitives]

IOB (I/O レジスタの IOB 内へのパック)

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。

IOB 制約が 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。

- ・ [Optimization Goal] が area に設定されている場合は、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。
- ・ [Optimization Goal] が speed に設定されている場合は、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。

PRIORITY_EXTRACT (プライオリティ エンコーダの抽出)

PRIORITY_EXTRACT を使用すると、プライオリティ エンコーダ マクロの推論を有効または無効にできます。

PRIORITY_EXTRACT に設定できる値は、次のとおりです。

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **force** (XCF のみ)

各エンコーダの記述に対し、内部決定ルールに従ってマクロが作成されるか、または残りのロジックと共に最適化されます。force に設定すると、このルールが無視され、常にマクロが作成されます。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、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、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 構文例

次のように宣言します。

```
attribute priority_extract: string;
```

次のように指定します。

```
attribute priority_extract of {signal_name | entity_name}: {signal |  
entity} is "{yes | no | force}";
```

デフォルトでは、yes に設定されています。

Verilog 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。

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" priority_extract={yes | no | true | false | force};
```

XST Constraint File (XCF) 構文例 2

```
BEGIN MODEL "entity_name" NET "signal_name" priority_extract={yes | no | true | false | force};END;
```

XST コマンド ライン

XST コマンド ラインで次のように定義します。

```
xst run -priority_extract {yes | no | force}
```

デフォルトでは、yes に設定されています。

ISE Design Suite からの設定

ISE® Design Suite でこの制約をグローバルに設定するには、次をクリックします。

[Process Properties] ダイアログ ボックス → [HDL Options] → [Priority Encoder Extraction]

RAM_EXTRACT (RAM の抽出)

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。

RAM_EXTRACT には、次の値を使用できます。

- **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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、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、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 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 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。

```
(* ram_extract = "{yes | no}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" ram_extract={yes | no | true | false};
```

XST Constraint File (XCF) 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

XST コマンドラインで次のように定義します。

```
xst run -ram_extract {yes | no}
```

デフォルトでは、yes に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [RAM Extraction]

RAM_STYLE (RAM スタイル)

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_STYLE には、次の値を使用できます。

- ・ **auto** (デフォルト)
- ・ **block**
- ・ **distributed**
- ・ **pipe_distributed**
- ・ **block_power1**
- ・ **block_power2**

デフォルトでは、auto に設定されています。

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 BRAM 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、VHDL、Verlog、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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

block_power1 および block_power2 は、Virtex®-4 と Virtex-5 デバイスでのみサポートされます。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、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、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 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}";
```

デフォルトでは、auto に設定されています。

Verilog 構文例

この制約は、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}" *)
```

デフォルトでは、auto に設定されています。

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" ram_style={auto | block | distributed | pipe_distributed  
| block_power1 | block_power2};
```

XST Constraint File (XCF) 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

XST コマンドラインで次のように定義します。

```
xst run -ram_style {auto | block | distributed}
```

デフォルトでは、auto に設定されています。

コマンドラインでは、pipe_distributed には設定できません。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [RAM Style]

REDUCE_CONTROL_SETS (制御セットの削減)

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。

REDUCE_CONTROL_SETS には、次の 2 つの値を使用できます。

- ・ **auto**
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®-5 デバイスにのみ適用できます。これ以外の FPGA には適用できません。CPLD には適用できません。

適用可能エレメント

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

XST コマンドラインで次のように定義します。

```
xst run -reduce_control_sets {auto | no}
```

デフォルトでは no に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Reduce Control Sets]

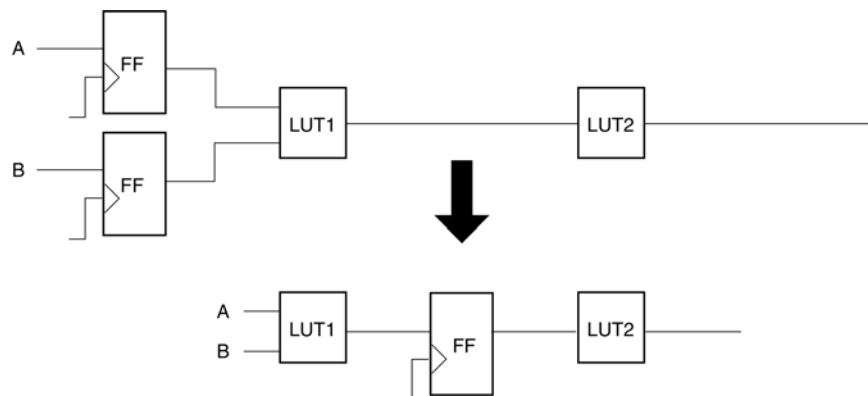
REGISTER_BALANCING (レジスタの自動調整)

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。

REGISTER_BALANCING には、次の 2 つのカテゴリがあります。

- ・ 順方向のレジスタ自動調整
- ・ 逆方向のレジスタ自動調整

順方向のレジスタ自動調整

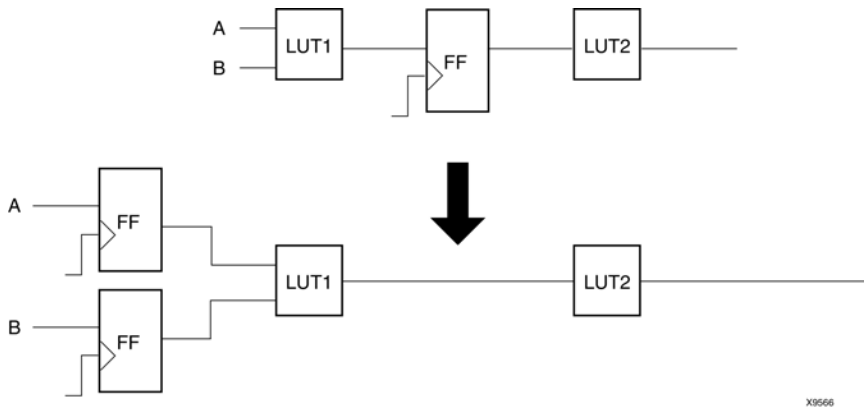


LUT の各入力にあるフリップフロップすべてを 1 つのフリップフロップとして LUT の出力に移動します。

複数のフリップフロップが 1 つのフリップフロップに置き換わる際には、次のように LUT 名に基づいた名前が選択されます。

`LutName_FRBId`

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。

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_BRBId`

設定できる値は次のいずれかです。

- ・ **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)

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)

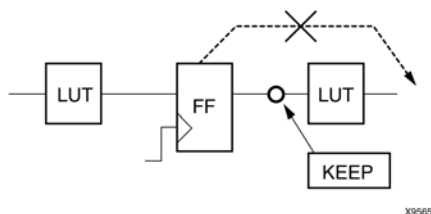
また、次の制約もレジスタ自動調整に影響を与えます。

- ・ 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。

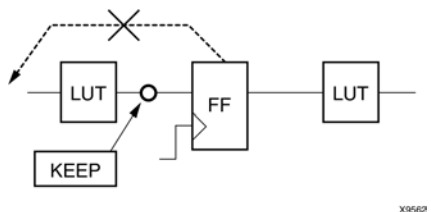
キープ (KEEP)

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。

入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お、REGISTER_BALANCE=NO と同じになります。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

次に適用されます。

- ・ デザイン全体 (XST コマンド ラインまたは ISE® Design Suite を使用)
- ・ 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、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 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 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}" *)
```

デフォルトでは no に設定されています。

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" register_balancing={yes | no | true | false | forward | backward};
```

XST Constraint File (XCF) 構文例 2

```
BEGIN MODEL "entity_name" NET "primary_clock_signal" register_balancing={yes | no |  
true | false | forward | backward};END;
```

XST Constraint File (XCF) 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

XST コマンドラインで次のように定義します。

```
xst run -register_balancing {yes | no | forward | backward}
```

デフォルトでは no に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Register Balancing]

REGISTER_DUPLICATION (レジスタの複製)

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。

REGISTER_DUPLICATION には、次の値を使用できます。

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

デフォルトでは、yes に設定されています。

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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

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 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 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}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" register_duplication={yes | no | true | false};
```

XST Constraint File (XCF) 構文例 2

```
BEGIN MODEL "entity_name"
```

```
NET "signal_name" register_duplication={yes | no | true | false};
```

```
END;
```

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Register Duplication]

ROM_EXTRACT (ROM の抽出)

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。

ROM_EXTRACT には、次の値を使用できます。

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

デフォルトでは、yes に設定されています。

ROM は通常、割り当てられた値がすべて定数である case 文から推論されます。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、またはデザイン エレメント、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、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 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 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}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" rom_extract={yes | no | true | false};
```

XST Constraint File (XCF) 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

XST コマンド ラインで次のように定義します。

```
xst run -rom_extract {yes | no}
```

デフォルトでは、yes に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [ROM Extraction]

ROM_STYLE (ROM スタイル)

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 を使用する場合は、まず [ROM の抽出 \(ROM_EXTRACT\)](#) を yes に設定しておく必要があります。

ROM_STYLE には、次の値を使用できます。

- ・ **auto** (デフォルト)
- ・ **block**

デフォルトでは、auto に設定されています。

推論された各 ROM に対して最適なインプリメント方法が自動設定されます。ほかの値を選択すると、ブロック ROM または分散 ROM を使用するようになります。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、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、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 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}";
```

デフォルトでは、auto に設定されています。

Verilog 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。

```
(* rom_style = "{auto | block | distributed}" *)
```

デフォルトでは、auto に設定されています。

XST Constraint File (XCF) 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};
```

XST Constraint File (XCF) 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

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。

XST コマンド ラインで次のように定義します。

```
xst run -rom_style {auto | block | distributed}
```

デフォルトでは、auto に設定されています。

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 で次のように定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [ROM Style]

SHREG_EXTRACT (シフトレジスタの抽出)

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。

SHREG_EXTRACT には、次の値を設定できます。

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

FPGA デバイスでこの SHREG_EXTRACT 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、「[シフトレジスタの Hardware Description Language \(HDL\) 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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、またはデザイン エレメント、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、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 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 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}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" shreg_extract={yes | no | true | false};
```

XST Constraint File (XCF) 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

XST コマンド ラインで次のように定義します。

```
xst run -shreg_extract {yes | no}
```

デフォルトでは、yes に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [Shift Register Extraction]

-slice_packing (スライス パッキング)

-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、重要な LUT 間の接続をスライスまたは CLB 内に配置します。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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

この値は、ISE® Design Suite の [Process Properties] ダイアログ ボックスの [Xilinx Specific Options] ページにある [Slice Packing] で設定します。

構文

```
-slice_packing {yes|no}
```

構文例

```
xst run -slice_packing no
```

XST の内部パック機能を無効にします。

USELOWSKEWLINES (ロー スキュー ラインの使用)

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。

XOR_COLLAPSE (XOR コラプス)

XOR_COLLAPSE を使用すると、カスケード接続された XOR を 1 つの XOR にまとめるかどうかを指定できます。

XOR_COLLAPSE には、次の値を設定できます。

- ・ **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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

カスケードされた XOR 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。

VHDL 構文例

次のように宣言します。

```
attribute xor_collapse: string;
```

次のように指定します。

```
attribute xor_collapse {signal_name | entity_name}: {signal | entity} is "{yes | no}";
```

デフォルトでは、yes に設定されています。

Verilog 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。

```
(* xor_collapse = "{yes | no}" *)
```

デフォルトでは、yes に設定されています。

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" xor_collapse={yes | no | true | false};
```

XST Constraint File (XCF) 構文例 2

```
BEGIN MODEL "entity_name"
```

```
NET "signal_name" xor_collapse={yes | no | true | false};
```

```
XOR Collapsing END;
```

XST コマンドライン

XST コマンドラインで次のように定義します。

```
xst run -xor_collapse {yes | no}
```

デフォルトでは、yes に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [XOR Collapsing]

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、次の合計の絶対値またはパーセントで指定します。

- ・ LUT と FF のペア (Virtex®-5 デバイス)
- ・ スライス (それ以外のデバイス)

このエリア制約を満たすことができない場合は、エリア制約を無視してタイミング最適化が実行されます。自動的にリソースが管理されないようにするには、-1 を指定します。詳細は、「[エリア制約を設定した場合のスピード最適化](#)」を参照してください。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

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 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#";
```

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。

Verilog 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#" *)
```

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。

XST Constraint File (XCF) 構文例

```
MODEL "entity_name" slice_utilization_ratio=integer;
```

```
MODEL "entity_name" slice_utilization_ratio=integer%;
```

```
MODEL "entity_name" slice_utilization_ratio=integer#;
```

整数値は最初の 2 つの例ではパーセントとして処理され、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。

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。

XST コマンド ライン

XST コマンド ラインで次のように定義します。

```
xst run -slice_utilization_ratio integer
xst run -slice_utilization_ratio integer%
xst run -slice_utilization_ratio integer#
```

整数値は最初の 2 つの例ではパーセントとして処理され、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 で次のように定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Slice Utilization Ratio] または [Process Properties] ダイアログ ボックス → [Synthesis Options] → [LUT-FF Pairs Utilization Ratio] をクリックします。

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) 使用率の許容範囲 (SLICE_UTILIZATION_RATIO)に関連する制約です。この制約では、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、「[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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

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 構文例

次のように宣言します。

```
attribute slice_utilization_ratio_maxmargin: string;
```

次のように指定します。

```
attribute slice_utilization_ratio_maxmargin of entity_name : entity is "integer";  
attribute slice_utilization_ratio_maxmargin of entity_name : entity is "integer%";  
attribute slice_utilization_ratio_maxmargin of entity_name : entity is "integer#";
```

整数値は最初の 2 つの例ではパーセントとして処理され、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 構文例

次をモジュール宣言またはインスタンス化の直前に入力します。

```
(* slice_utilization_ratio_maxmargin = "integer" *)  
(* slice_utilization_ratio_maxmargin = "integer%" *)  
(* slice_utilization_ratio_maxmargin = "integer#" *)
```

整数値は最初の 2 つの例ではパーセントとして処理され、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。

XST Constraint File (XCF) 構文例

```
MODEL "entity_name" slice_utilization_ratio_maxmargin=integer;  
MODEL "entity_name" slice_utilization_ratio_maxmargin="integer%";  
MODEL "entity_name" slice_utilization_ratio_maxmargin="integer#";
```

整数値は最初の 2 つの例ではパーセントとして処理され、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。

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。

XST コマンドライン

XST コマンドラインで次のように定義します。

```
xst run -slice_utilization_ratio_maxmargin integer  
xst run -slice_utilization_ratio_maxmargin integer%  
xst run -slice_utilization_ratio_maxmargin integer#
```

整数値は最初の 2 つの例ではパーセントとして処理され、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。

LUT_MAP (単一 LUT へのエンティティのマップ)

LUT_MAP を使用すると、1 つのブロックを 1 つの LUT にマップするように指定できます。RTL レベルで記述された機能が 1 つの LUT にフィットしない場合は、エラー メッセージが表示されます。

UNISIM ライブラリを使用すると、LUT コンポーネントを直接 HDL コードにインスタンス化できます。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。

INIT でファンクションを定義するのが不都合な場合は、ファンクションを個別のブロックとして VHDL または Verilog で記述し、LUT にマップする方法もあります。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、「INIT および RLOC の指定」を参照してください。

XST では、Synplicity でサポートされる 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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

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 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 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}" *)
```

XST Constraint File (XCF) 構文例

```
MODEL "entity_name" lut_map={yes | no | true | false};
```

USE_CARRY_CHAIN (キャリーチェーンの使用)

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よ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。USE_CARRY_CHAIN はグローバルにも、またはローカルにも設定できます。

USE_CARRY_CHAIN には、次の値を設定できます。

- ・ **yes** (デフォルト)
- ・ **no**

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体または信号に適用されます。

適用ルール

設定された信号に適用されます。

構文例

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。

回路図からの設定

- ・ 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 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 構文例

次を信号宣言の直前に入力します。

```
(* use_carry_chain = "{yes | no}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" use_carry_chain={yes | no | true | false}X;
```

XST Constraint File (XCF) 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

XST コマンドラインで次のように定義します。

```
xst run -use_carry_chain {yes | no}
```

デフォルトでは、yes に設定されています。

TRISTATE2LOGIC (トライステートからロジックへの変換)

デバイスによっては内部トライステートがサポートされないの、トライステートが自動的に等価ロジックに変換されます。トライ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、内部トライステートをロジックに変換すると、スピードを向上できます。場合によっては、エリア最適化の結果が向上することもあります。通常はトライステートをロジックに変換するとエリアが増加します。OPT_MODE 制約を area に設定している場合は、TRISTATE2LOGIC (Convert Tristates to Logic) を no に設定する必要があります。

次のような値を設定できます。

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

TRISTATE2LOGIC 制約には、次のような制限があります。

- ・ ロジックに変換されるのは、内部トライステートのみです。出力パッドに接続された最上位モジュールのトライステートは保持されます。
- ・ Spartan®-3 または Virtex®-4 デバイスのような内部トライステートを持たないアーキテクチャには、適用できません。これらのアーキテクチャでは、自動的にトライステートがロジックに変換されます。階層が保持されていたり、ブロックごとに合成を実行するなどしてデザイン全体が認識されていない場合など、不正なビヘイビアやマルチソースを招くことになると XST で判断される場合は、自動的に変換されないことがあります。このような場合、下位の最適化段階で警告メッセージが出力されます。デザインによっては、デザインフローを続行して MAP でロジックを変換するか、または特定ブロックまたは信号に TRISTATE2LOGIC=YES を設定して強制的にロジックに変換することができる場合もあります。
- ・ 次の場合、XST でトライステートがロジックに変換されません。
 - － トライステートがブラックボックスに接続されている
 - － トライ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
 - － トライステートが最上位レベルの出力に接続されている
 - － トライステートが配置されたブロックまたはトライステートが接続された信号で TRISTATE2LOGIC が no に設定されている

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

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、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 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 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}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" tristate2logic={yes|no|true|false};
```

XST Constraint File (XCF) 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

XST コマンドラインで次のように定義します。

```
run -tristate2logic {yes|no}
```

デフォルトでは、yes に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Convert Tristates to Logic]

USE_CLOCK_ENABLE (クロック イネーブルの使用)

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 のプロトタイプを FPGA で作成する場合は、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。フリップフロップをインスタンス化すると、OPTIMIZE_PRIMITIVE 制約が yes に設定されている場合にのみ、CE が削除されます。

USE_CLOCK_ENABLE の値には、次を設定できます。

- ・ **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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

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、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 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 構文例

次インスタンス、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}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" use_clock_enable={auto | yes | no | true | false};
```

XST Constraint File (XCF) 構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" use_clock_enable={auto | yes | no | true | false};  
END
```

XST Constraint File (XCF) 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

XST コマンド ラインで次のように定義します。

```
xst run -use_clock_enable {auto | yes | no}
```

デフォルトでは、auto に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Use Clock Enable]

USE_SYNC_SET (同期セットの使用)

USE_SYNC_SET を使用すると、フリップフロップの同期セットの使用を有効または無効にできます。ASIC のプロトタイプを FPGA で作成する場合は、通常同期セット機能を無効にします。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同期セットリソースが使用されません。また、デザインによっては、フリップフロップのデータ入力に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。フリップフロップをインスタンス化すると、OPTIMIZE_PRIMITIVE 制約が yes に設定されている場合にのみ、同期リセットが削除されます。

設定できる値は次のいずれかです。

- ・ **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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

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、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 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 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。

```
(* use_sync_set = "{auto | yes | no}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" use_sync_set={auto | yes | no | true | false};
```

XST Constraint File (XCF) 構文例 2

```
BEGIN MODEL "entity_name"
```

```
NET "signal_name" use_sync_set={auto | yes | no | true | false};
```

```
END;
```

XST Constraint File (XCF) 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

XST コマンド ラインで次のように定義します。

```
xst run -use_sync_set {auto | yes | no}
```

デフォルトでは、auto に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Use Synchronous Set]

USE_SYNC_RESET (同期リセットの使用)

USE_SYNC_RESET を使用すると、フリップフロップの同期リセットの使用を有効または無効にできます。ASIC のプロトタイプを FPGA で作成する場合は、通常同期セット機能を無効にします。

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同期セットリソースが使用されません。また、デザインによっては、フリップフロップのデータ入力に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。フリップフロップをインスタンス化すると、OPTIMIZE_PRIMITIVE 制約が yes に設定されている場合にのみ、同期リセットが削除されます。

設定できる値は次のいずれかです。

- ・ **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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

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、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 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}:{entity  
| component | signal | label} is "{auto | yes | no}";
```

Verilog 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。

```
(* use_sync_reset = "{auto | yes | no}" *)
```

USE_SYNC_RESET (Use Synchronous Reset) XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" use_sync_reset={auto | yes | no | true | false};
```

XST Constraint File (XCF) 構文例 2

```
BEGIN MODEL "entity_name "  
NET "signal_name" use_sync_reset={auto | yes | no | true | false};  
END;
```

XST Constraint File (XCF) 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

XST コマンドラインで次のように定義します。

```
xst run -use_sync_reset {auto | yes | no}
```

デフォルトでは、auto に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Use Synchronous Set]

USE_DSP48 (DSP48 の使用)

この制約は、次のように呼ばれます。

- ・ Use DSP48 (Virtex®-4 デバイス)
- ・ Use DSP Block (Virtex-5 および Spartan®-3A DSP デバイス)

Virtex-4 以降のデバイスに含まれる DSP48 ブロック リソースの使用を有効にします。

デフォルトでは、auto に設定されています。

デフォルト値の auto を使用した場合、MAC などのマクロは DSP48 に自動的にインプリメントされますが、加算器など一部のマクロはスライスにインプリメントされます。これらのマクロを DSP48 にインプリメントするには、USE_DSP48 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 DSP48 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、DSP48 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、最初のレジスタ段を DSP48 の外にインプリメントするには、出力に [キープ \(KEEP\)](#) 制約を設定する必要があります。

設定できる値は次のいずれかです。

- ・ **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 DSP48 リソースの数が制御されます。デフォルトでは、すべての使用可能な DSP48 リソースが可能な限り使用されます。詳細は、「**DSP48 ブロック リソース**」を参照してください。

アーキテクチャ サポート

次の FPGA デバイスでのみ使用できます。これ以外の FPGA には適用できません。CPLD には適用できません。

- ・ Spartan-3A DSP
- ・ Virtex-4
- ・ Virtex-5

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 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 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。

```
(* use_dsp48 = "{auto | yes | no}" *)
```

XST Constraint File (XCF) 構文例 1

```
MODEL "entity_name" use_dsp48={auto | yes | no | true | false};
```

XST Constraint File (XCF) 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

XST コマンド ラインで次のように定義します。

```
-use_dsp48 {auto|yes|no}
```

デフォルトでは、auto に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Process Properties] ダイアログ ボックス → [HDL Options] → [Use DSP48]

CPLD 制約 (タイミング以外)

このセクションでは、CPLD の制約 (タイミング以外) について説明します。これは、FPGA には適用されません。

- ・ クロック イネーブル (`-pld_ce`)
- ・ データ ゲート (`DATA_GATE`)
- ・ マクロの保持 (`-pld_mp`)
- ・ 削減なし (`NOREDUCE`)
- ・ WYSIWYG (`-wysiwyg`)
- ・ XOR の保持 (`-pld_xp`)

`-pld_ce` (クロック イネーブル)

`-pld_ce` (クロック イネーブル) では、順序ロジックにクロック イネーブルが含まれる場合のインプリメント方法を指定します。専用のデバイスリソースを使用するか、または等価ロジックを生成するかを指定できます。

次のいずれかの値を選択します。

- ・ **yes**
デバイスのクロック イネーブル信号を使用してインプリメントします。
- ・ **no**
等価ロジックを使用してインプリメントします

クロック イネーブル信号を使用するかしないかは、デザイン ロジックによって判断します。たとえば、クロック イネーブルがブール代数式の結果である場合、フリップフロップの入力データはクロック イネーブル表現と結合すると簡略化されるため、このオプションを `no` に設定した方がフィットの結果が向上する可能性があります。

アーキテクチャ サポート

すべての CPLD デバイスに適用されます。FPGA には適用できません。

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。

ISE® Design Suite の [Process Properties] ダイアログ ボックスの [Xilinx Specific Options] ページにある [Clock Enable] でグローバルに設定します。

構文

```
xst run -pld_ce {yes|no}
```

デフォルトでは、`yes` に設定されています。

構文例

```
xst run -pld_ce yes
```

クロック イネーブルをグローバルに yes に定義すると、クロック イネーブル ファンクションが同等のロジック全体にインプリメントされます

DATA_GATE (データ ゲート)

この制約を使用すると、デザインの消費電力を低減することができます。入力信号の遷移が CPLD デザインのファンクションに関係ない場合に、信号の遷移が伝搬されないようブロックするラッチが使用できるようになります。

入力が遷移すると、その遷移がデザインのファンクションに影響しない場合でも、CPLD のファンクション ブロックに配線されているため、電力を消費します。DataGate ラッチ ライブラリ プリミティブを I/O ピンの入力に接続することにより、このラッチのイネーブル ピンをアサートしたときに、信号の遷移がブロックされ、電力が消費されないようにすることができます。

詳細については、『[制約ガイド](#)』を参照してください。

アーキテクチャ サポート

この制約は、CoolRunner™-II デバイスにのみ使用できます。

-pld_mp (マクロの保持)

-pld_mp を使用すると、デザイン階層の処理とは別に、マクロを処理できるように指定できます。この制約を設定すると、マクロを階層モジュールとして保持しながら、すべての階層ブロックを最上位モジュールに結合できます。周辺のロジックと結合されるマクロを除き、デザイン階層も保持できます。マクロを周辺ロジックと結合した方が、デザインのフィットで良い結果が得られる場合があります。

次のいずれかの値を選択します。

- ・ **yes**
マクロが保持され、マクロ生成機能により生成されます。
- ・ **no**
マクロが保持されず、HDL 合成機能により生成されます。

[Flatten Hierarchy] の値によって、保持されないマクロは、デザイン ロジックに結合されるか、階層ブロックになります。

[Flatten Hierarchy] の値	処理
yes ()	デザイン ロジックに結合されます。
no ()	階層ブロックになります。

2 ビット加算器、4 ビット マルチプレクサなどの小型のマクロは、[Macro Preserve] および [Flatten Hierarchy] の設定に関係なく、常に周辺ロジックと結合されます。

アーキテクチャ サポート

すべての CPLD デバイスに適用されます。FPGA には適用できません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

この値は、ISE® Design Suite の [Process Properties] ダイアログ ボックスの [Xilinx Specific Options] ページにある [Macro Preserve] で設定します。

構文

```
xst run -pld_mp {yes|no}
```

デフォルトでは、yes に設定されています。

構文例

```
xst run -pld_mp no
```

マクロは保持されず、HDL 合成機能により生成されます。

NOREDUCED (削減なし)

NOREDUCED (削減なし) を使用すると、次が実行できます。

- ・ ロジック ハザードやレース コンディションを避けるためにデザインに含まれている冗長な論理記述が最小化されないように指定します。
- ・ 適正なマップが実行されるように組み合わせフィードバック ループの出力ノードが識別されます。

詳細は、『[制約ガイド](#)』の「NOREDUCED」を参照してください。

-wysiwyg (WYSIWYG)

-wysiwyg (WYSIWYG) を使用すると、ユーザー指定を最大限に反映させたネットリストが作成できます。つまり、Hardware Description Language (HDL) デザインで宣言されたすべてのノードが保持されます。

yes に設定すると、XST では次が実行されます。

- ・ すべてのユーザー内部信号 (ノード) が保持されます。
- ・ NGC ファイルにこれらのノードすべてに対して SOURCE_NODE 制約が作成されます。
- ・ コラプス、因数分解などのデザイン最適化は実行されません。

ブール代数式の最小化のみが実行されます

アーキテクチャ サポート

すべての CPLD デバイスに適用されます。FPGA には適用できません。

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

XST コマンドラインで次のように定義します。

```
xst run -wysiwyg {yes | no}
```

デフォルトでは no に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックス → [Synthesis Options] → [WYSIWYG]

-pld_xp (XOR の保持)

-pld_xp を使用すると、XOR マクロ階層のフラット化を有効または無効にできます。

CPLD フローでは、Hardware Description Language (HDL) 合成で推論された XOR もマクロ ブロックと見なされますが、デバイスのマクロセルの XOR ゲートをより柔軟に使用できるようにするため、別に処理されます。このため [Flatten Hierarchy] をオンにし [Macro Preserve] をオフにして、デザインをフラットにしながら、XOR ゲートを維持できます。XOR を保持すると、デザインを大幅に簡略化できます。

次のいずれかの値を選択します。

- ・ **yes** (デフォルト)
XOR マクロが保持されます。
- ・ **no** ()
XOR マクロは周辺ロジックと結合されます。

通常、XOR マクロを保持すると積項の数が低減され、良い結果が得られます。完全にフラット化されたネットリストを生成する場合は、no に設定します。no は、完全にフラットなネットリストが必要な場合に使用してください。完全にフラット化されたデザインでグローバル最適化を実行することにより、デザインのフィットが向上する場合があります。

次のように設定すると、完全にフラット化されたデザインが生成されます。

- ・ [Flatten Hierarchy]
yes ()
- ・ [Macro Preserve] オプション
no ()
- ・ XOR 保持
no ()

ただし、no にしていても、必ずしも XOR 演算子が Electronic Data Interchange Format (EDIF) ネットリストから削除されるわけではありません。ネットリスト生成の段階では、ロジックを簡略化するため、XOR ゲートの推論が試みられます。この処理は、HDL 合成段階で行われる XOR の保持とは関係なく、ロジックを簡略化する目的で行われます。

ISE® Design Suite の [Process Properties] ダイアログ ボックスの [Xilinx Specific Options] ページにある [XOR Preserve] で設定します。

アーキテクチャ サポート

すべての CPLD デバイスに適用されます。FPGA には適用できません。

適用可能エレメント

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

```
xst run -pld_xp {yes|no}
```

デフォルトでは、yes に設定されています。

構文例

```
xst run -pld_xp no
```

XOR マクロは周辺ロジックと結合されます。

タイミング制約

このセクションでは、XST のタイミング制約を適用する方法と特定の制約について説明します。

タイミング制約の指定

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)
- ・ ISE® Design Suite : [Process] → [Process Properties] → [Synthesis Options] → [Global Optimization Goal]
- ・ User Constraints File (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 制約 \(FROM-TO\)](#)

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

タイミング制約は、デフォルトのままでは 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。

- ・ 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
- ・ コマンドラインの場合は、`-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に次の 3 つがあります。

- ・ 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`)

`-cross_clock_analysis` (クロス クロック解析)

`-cross_clock_analysis` を使用すると、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できます。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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

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 [Cross Clock Analysis] で設定します。

構文

```
xst run -cross_clock_analysis {yes|no}
```

構文例

```
xst run -cross_clock_analysis yes
```

タイミングの最適化中に複数のクロックドメイン間の解析が実行されます。

`-write_timing_constraints` (タイミング制約の書き込み)

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
- ・ コマンドラインの場合は、`-write_timing_constraints` コマンドライン オプションを使用

タイミング制約は、デフォルトのままでは 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

アーキテクチャに依存しません。

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

XST コマンドラインで次のように定義します。

```
xst run -write_timing_constraints {yes | no}
```

デフォルトでは、yes に設定されています。

ISE Design Suite からの設定

ISE® Design Suite で次のように定義します。

[Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックス → [Synthesis Options] → [Write Timing Constraints]

CLOCK_SIGNAL (クロック信号)

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。CLOCK_SIGNAL 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

すべての FPGA デバイスに適用されます。CPLD には適用できません。

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 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 構文例

次を信号宣言の直前に入力します。

```
(* clock_signal = "{yes | no}" *)
```

XST Constraint File (XCF) 構文例

```
BEGIN MODEL "entity_name" NET "primary_clock_signal" clock_signal={yes | no | true | false};END;
```

-glob_opt (グローバルな最適化目標)

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

-glob_opt を使用すると、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、「[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** (register to register) は、デザイン全体の周期を最適化します。
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。これがデフォルト設定です。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** (inpad to register) は、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。
XST では、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からプライマリ出力ポートまでのパスがすべて識別されます。
- ・ **OFFSET_OUT_AFTER** (register to outpad) は、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。
XST では、すべての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すべて識別されます。
- ・ **INPAD_TO_OUTPAD** (inpad to outpad) は、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** は、上記すべての制約をまとめた制約です。

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。

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。

構文

-glob_opt

```
{allclocknets|offset_in_before|offset_out_after|inpad_to_outpad|max_delay}
```

構文例

```
xst run -glob_opt OFFSET_OUT_AFTER
```

デザイン全体の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。

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。

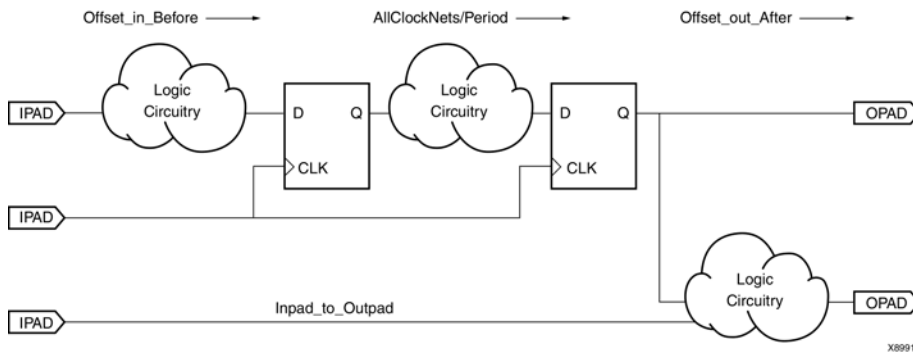
- ALLCLOCKNETS (レジスタ間) : デザイン内のすべてのクロックに対し、同じクロックパス上にあるレジスタ間のすべてのパスがデフォルトで指定されます。クロックドメイン遅延を考慮に入れるには、クロスクロック解析 (-cross_clock_analysis) オプションを yes に設定します。
- OFFSET_IN_BEFORE (入力パッドとレジスタ間) プライマリ入力ポートからすべての順序エレメント、または指定したクロック信号名で駆動される特定の順序エレメントまでのパスを識別します。
- OFFSET_OUT_AFTER (レジスタと出力パッド間) は OFFSET_IN_BEFORE に類似していますが、順序エレメントからすべてのプライマリ出力ポートまでのパスに対して制約を指定します。
- INPAD_TO_OUTPAD (入力パッドから出力パッド) : 最大組み合わせパスに対して制約が指定されます。
- MAX_DELAY : 次のタイミング制約で定義されるすべてのパスが指定されます。

ALLCLOCKNETS

OFFSET_IN_BEFORE

OFFSET_OUT_AFTER

INPAD_TO_OUTPAD



XST Constraint File (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。詳細は、[階層区切り文字の指定 \(-hierarchy_separator\)](#) 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] 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。

XST 制約ファイル (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 制約 (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)

PERIOD (周期)

PERIOD は、基本的なタイミング制約および合成制約です。PERIOD 制約を指定すると、デスティネーション エLEMENT のグループで定義されているクロックドメイン内で、すべての同期ELEMENT間のタイミングが確認されます。クロックが別のクロックの関数として定義されている場合、グループには複数のクロックドメインを通過するパスが含まれます。詳細は、『[制約ガイド](#)』の「PERIOD」を参照してください。

XST Constraint File (XCF) 構文例

```
NET netname PERIOD = value [{HIGH | LOW} value];
```

OFFSET (オフセット)

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。

OFFSET 制約を設定すると、次のような処理が可能になります。

- ・ 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。

構文例

```
OFFSET = {IN | OUT} offset_time [units] {BEFORE | AFTER} clk_name[ TIMEGRP group_name ];
```

FROM-TO (From-To 制約)

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 (FFS、PADS、RAMS) です。詳細は、『[制約ガイド](#)』の「FROM-TO」を参照してください。

XST Constraint File (XCF) 構文例

```
TIMESPEC TSname = FROM group1 TO group2 value;
```

TNM (タイミング名)

TNM は、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。この制約を使用すると、特定の FFS、RAMS、LATCHES、PADS、BRAMS_PORTA、BRAMS_PORTB、CPUS、HSIOS、MULTS をグループのELEMENTとして指定し、タイミング仕様の適用を簡略化できます。

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。

構文例

```
{INST | NET | PIN} inst_net_or_pin_name TNM = [predefined_group:] identifier;
```

TNM_NET (ネットのタイミング名)

TNM_NET は、入力パッド ネットに設定する場合を除き、ネットに設定した TNM と基本的に同等です。DLL/DCM に、PERIOD 制約を使用した TNM_NET を設定する場合は、特別なルールが適用されます。詳細は、『[制約ガイド](#)』の「CLKDLL および DCM での PERIOD 指定」を参照してください。

TNM_NET は通常、特定ネットを指定するため Hardware Description Language (HDL) デザインで使用するプロパティです。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。

構文例

```
NET netname TNM_NET = [predefined_group:] identifier;
```

TIMEGRP (タイムグループ)

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、ほかのグループを基にしてグループを定義できます。TIMEGRP 制約を使用すると、既存グループを組み合わせて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 制約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。

構文例

```
TIMEGRP newgroup = existing_grp1 existing_grp2 [existing_grp3 ...];
```

TIG (タイミング無視)

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。

XST Constraint File (XCF) 構文例

```
NET net_name TIG;
```

インプリメンテーション制約

インプリメンテーション制約は、配置および配線を制御します。これらの制約は XST では直接使用されず、インプリメンテーション ツールに渡されて使用されます。また、インプリメンテーション制約が設定されたオブジェクトは保持されます。

インプリメンテーション制約と同等のバイナリ コードが NGC ファイルに記述されます。ファイルはバイナリなので、NGC ファイルでインプリメンテーション制約を編集することはできません。

XST Constraint File (XCF) のインプリメンテーション制約のコード記述については、『[インプリメンテーション制約の構文例](#)』を参照してください。

詳細は、『[制約ガイド](#)』を参照してください。

インプリメンテーション制約の構文例

このセクションでは、インプリメンテーション制約の構文例を示します。

- ・ インプリメンテーション制約の XST Constraint File (XCF) 構文例
- ・ インプリメンテーション制約の VHDL 構文例
- ・ インプリメンテーション制約の Verilog 構文例

インプリメンテーション制約の XST Constraint File (XCF) 構文例

制約をエンティティ全体に適用するには、次のいずれかの XST Constraint File (XCF) 構文を使用します。

```
MODEL EntityName PropertyName;  
MODEL EntityName PropertyName =PropertyValue;
```

エンティティ内の特定のインスタンス、ネット、またはピンに制約を適用するには、次のいずれかの構文を使用します。

```
BEGIN MODEL EntityName {NET | INST | PIN} {NetName | InstName | SigName} PropertyName; END;  
BEGIN MODEL EntityName {NET | INST | PIN} {NetName | InstName |  
SigName} PropertyName=PropertyValue; END;
```

インプリメンテーション制約の VHDL 構文例

VHDL の場合、次のように指定する必要があります。

```
attribute PropertyName of {NetName | InstName | PinName} : {signal | label} is "PropertyValue";
```

インプリメンテーション制約の Verilog 構文例

Verilog の場合、次のように指定する必要があります。

```
// synthesis attribute PropertyName of {NetName | InstName | PinName} is "PropertyValue";
```

Verilog-2001 の場合、参照する信号、モジュール、またはインスタンスの前で次のように指定する必要があります。

```
(* PropertyName="PropertyValue" *)
```

RLOC

すべての FPGA デバイスに適用されます。CPLD には適用できません。

FPGA デバイスをターゲットとしている場合、RLOC 制約を使用して、FPGA チップ上でのデザイン エLEMENT の配置をほかのELEMENTに相対的に指定できます。たとえば、sr11 という名前の SRL16 インスタンスを R9C0.S0 に配置する場合、Verilog コードで次のように指定します。

```
// synthesis attribute RLOC of sr11 : "R9C0.S0";
```

同じ属性は XST Constraint File (XCF) では次のように指定できます。

```
BEGIN MODEL ENTNAME  
INST sr11 RLOC=R9C0.S0;  
END;
```

このように指定すると、次の文と同等のバイナリコードが NGC ファイルに記述されます。

```
INST sr11 RLOC=R9C0.S0;
```

詳細は、『[制約ガイド](#)』の「RLOC」を参照してください。

NOREDUCE

すべての CPLD デバイスに適用されます。FPGA には適用できません。

指定した信号を生成するブール代数式が最適化されないように設定します。たとえば、ローカル信号を次のファンクションに割り当て、NOREDUCE 制約を信号 s に設定する場合、次のように指定します。

```
signal s : std_logic;  
attribute NOREDUCE : boolean;  
attribute NOREDUCE of s : signal is "true";  
...  
s <= a or (a and b);
```

XST Constraint File (XCF) で NOREDUCE を次のように指定します。

```
BEGIN MODEL ENTNAME  
  NET s NOREDUCE;  
  NET s KEEP;  
END;
```

この場合、XST は次の文を NGC ファイルに記述します。

```
NET s NOREDUCE;  
NET s KEEP;
```

詳細は、『[制約ガイド](#)』の「NOREDUCE」を参照してください。

PWR_MODE (電力モード)

PWR_MODE を使用すると、マクロセルの電力消費特性を制御できます。VHDL で次のように指定すると、信号 s を生成するファンクションで消費電力が低減されるように最適化されます。

```
attribute PWR_MODE : string;  
attribute PWR_MODE of s : signal is "LOW";
```

XST Constraint File (XCF) ファイルの場合、同じ制約を次のように指定します。

```
MODEL ENTNAME  
  NET s PWR_MODE=LOW;  
  NET s KEEP;  
END;
```

この場合、XST は次の文を NGC ファイルに記述します。

```
NET s PWR_MODE=LOW;  
NET s KEEP;
```

次の場合、XST では Hardware Description Language (HDL) 属性が信号に適用され、インスタンスが推論されます。

- ・ 属性がインスタンス (IOB、DRIVE、IOSTANDARD など) に適用されている場合
- ・ インスタンスが HDLソースにない (インスタンス化されていない) 場合

アーキテクチャ サポート

すべての CPLD デバイスに適用されます。FPGA には適用できません。

サードパーティの制約

このセクションでは、XST でサポートされるサードパーティの合成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

このセクションでは、制約とその制約と同等の XST 制約を示します。各制約の機能などについては、該当するベンダーのマニュアルを参照してください。

サードパーティ制約の中には、XST で自動的にサポートされるものがあります。次の表で「あり」になっている制約は、完全にサポートされています。部分的にのみサポートされる場合は、その詳細が次の表の自動識別の列に記述されています。

次の規則が適用されます。

- ・ 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

name	ベンダー名	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 Synopsys	フル ケース	なし	Verilog
ispad	Synplicity	なし	なし	なし
map_to_module	Synopsys	なし	なし	なし
net_name	Synopsys	なし	なし	なし
parallel_case	Synplicity Synopsys	パラレル ケース	なし	Verilog
return_port_name	Synopsys	なし	なし	なし
resource_sharing directives	Synopsys	リソース共有	なし	VHDL、Verilog
set_dont_touch_network	Synopsys	必要なし	なし	なし
set_dont_touch	Synopsys	必要なし	なし	なし
set_dont_use_cel_name	Synopsys	必要なし	なし	なし
set_prefer	Synopsys	なし	なし	なし
state_vector	Synopsys	なし	なし	なし
syn_allow_retiming	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

name	ベンダー名	同等の XST 制約	自動認識	HDL
syn_hier	Synplicity	階層の維持	あり syn_hier = hardrecognized askeep_hierarchy = soft syn_hier = removerecognized askeep_hierarchy = no (XST では、syn_hier の hard と remove 値しか 自動認識されません)	VHDL、Verilog
syn_isclock	Synplicity	なし	なし	なし
syn_keep	Synplicity	キープ	あり (最終ネットリストに 指定したネットは保持さ れますが、KEEP 制約 は付けません)	VHDL、Verilog
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	インスタンスシートされ 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_extract and ram_style	あり 指定してもしなくても no_rw_check モードでイ ンプリメントします。 area 値は無視されま す。	VHDL、Verilog
syn_reference_clock	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_extract および rom_style	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 <n>	Synplicity	なし	なし	なし
syn_tpd <n>	Synplicity	なし	なし	なし

name	ベンダー名	同等の XST 制約	自動認識	HDL
syn_tristate	Synplicity	なし	なし	なし
syn_tristatetomux	Synplicity	なし	なし	なし
syn_tsu <n>	Synplicity	なし	なし	なし
syn_useenables	Synplicity	なし	なし	なし
syn_useioff	Synplicity	I/O レジスタの IOB 内 へのバック	なし	VHDL、Verilog
synthesis translate_off	Synplicity Synopsys	変換なし	あり	VHDL、Verilog
synthesis translate_on		変換あり		
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 へのバック	なし	VHDL、Verilog
xc_isgsr	Synplicity	なし	なし	なし
xc_loc	Synplicity	LOC	あり	VHDL、Verilog
xc_map	Synplicity	LUT_MAP	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
- ・ XST Constraint File (XCF) 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
```

XST Constraint File (XCF) 構文例

キープ (KEEP) 制約は、合成制約ファイルでも設定できます。

```
BEGIN MODEL testkeep
  NET aux1 KEEP=true;
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。

VHDL 言語のサポート

この章では、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。この章は、次のセクションから構成されています。

- ・ [VHDL の IEEE サポート](#)
- ・ [サポートされる VHDL ファイル タイプ](#)
- ・ [VHDL で書き込み関数を使用したデバッグ](#)
- ・ [VHDL のデータ型](#)
- ・ [VHDL のレコード型](#)
- ・ [VHDL の初期値](#)
- ・ [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 の順序回路](#)
- ・ [VHDL の関数とプロシージャ](#)
- ・ [VHDL のアサート文](#)
- ・ [パッケージ文を使用した VHDL モデルの定義](#)
- ・ [サポートされる 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』
- ・ [デザイン制約](#)
- ・ [VHDL 属性の構文](#)

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次のようなハードウェア記述言語です。

- ・ 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や製造の前にシステム デザインをシミュレーションできます。ハードウェア プロトタイプに時間や費用を費やすことなく、正確性を検証できます。
- ・ 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 では、次がサポートされます。

- ・ VHDL IEEE 規格 1076-1987
- ・ VHDL IEEE 規格 1076-1993
- ・ VHDL IEEE 規格 1076-2006 (一部)

XST では、次の場合に VHDL IEEE 規格 1076-2006 がインスタンス化できます。

- ・ フォーマル ポートが bufe で、関連する実際のポートが out の場合
- ・ フォーマル ポートが out で、関連する実際のポートが buffer の場合

VHDL の IEEE の競合

VHDL IEEE 規格 1076-1987 は、VHDL IEEE 規格 1076-1993 と競合しない場合には使用できますが、競合する場合は、VHDL IEEE 規格 1076-1993 のビヘイビアが VHDL IEEE 規格 1076-1987 のビヘイビアを上書きします。

上書きされるのは次の場合です。

- ・ VHDL IEEE 規格 1076-1993 ではエラーになる
- ・ VHDL IEEE 規格 1076-1987 では問題がない

この場合、XST ではエラー メッセージではなく警告メッセージが表示されます (エラー メッセージの場合は、解析が停止されます)。

VHDL の IEEE の競合例

次は、VHDL の IEEE 競合の具体的な例です。

- ・ VHDL IEEE 規格 1076-1993 では、関数の宣言中に impure キーワードを使用するために impure 関数が必要
- ・ VHDL IEEE 規格 1076-1987 にそのような条件がない

この場合、XST は次のように動作します。

- ・ VHDL IEEE 規格 1076-1987 の VHDL コードを受諾
- ・ VHDL IEEE 規格 1076-1993 ビヘイビアに関する警告メッセージを表示

VHDL で LRM に準拠しない構文

XST では、LRM に準拠しない構文も一部サポートされます。サポートされるのは、次の場合です。

- ・ 制約が合成またはシミュレーション用のサードパーティ ツールのほとんどでサポートされる場合
- ・ コードに使用する言語に限界があり、結果や問題検出に影響のない場合

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 の場合 (またはその逆)、インスタンス化できません。

サポートされる VHDL ファイル タイプ

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。
- ・ ファイル書き込み機能はデバッグあるいは特定の値またはジェネリック値を外部ファイルに書き込むために使用できます。

詳細は、「[RAM の初期化のコード例](#)」を参照してください。

次の表の read 関数のいずれかを使用してください。これらの read 関数は次のパッケージでサポートされます。

- ・ **standard**
- ・ **std.textio**
- ・ **ieee.std_logic_textio**

機能	パッケージ ([Package])
file (text タイプのみ)	standard
access (line タイプのみ)	standard
file_open (file, name, open_kind)	standard
file_close (file)	standard
endfile (file)	standard
text	std.textio
line	std.textio
width	std.textio
readline (text, line)	std.textio
readline (line, bit, boolean)	std.textio
read (line, bit)	std.textio
readline (line, bit_vector, boolean)	std.textio
read (line, bit_vector)	std.textio
read (line, boolean, boolean)	std.textio
read (line, boolean)	std.textio
read (line, character, boolean)	std.textio
read (line, character)	std.textio
read (line, string, boolean)	std.textio
read (line, string)	std.textio
write (file, line)	std.textio
write (line, bit, boolean)	std.textio
write (line, bit)	std.textio
write (line, bit_vector, boolean)	std.textio
write (line, bit_vector)	std.textio
write (line, boolean, boolean)	std.textio
write (line, boolean)	std.textio
write (line, character, boolean)	std.textio
write (line, character)	std.textio
write (line, integer, boolean)	std.textio
write (line, integer)	std.textio
write (line, string, boolean)	std.textio

機能	パッケージ ([Package])
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

書き込み関数を使用した VHDL のデバッグ コード例

コード例は、本書が作成された時点のものです。アップデートは、
ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

書き込み関数を使用した VHDL のデバッグ コード例

```
--
-- Print 2 constants to the output file
--

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

entity file_support_1 is
    generic (data_width: integer:= 4);
    port( clk, sel: in std_logic;
          din: in std_logic_vector (data_width - 1 downto 0);
          dout: out std_logic_vector (data_width - 1 downto 0));
end file_support_1;

architecture Behavioral of file_support_1 is
    file results : text is out "test.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 + "1000";
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 (clk'event and clk='1') then
            if (sel = '1') then
                dout <= new_const;
            else
                dout <= din;
            end if;
        end if;
    end process;

end Behavioral;
```

VHDL で書き込み関数を使用してデバッグする場合のルール

VHDL で書き込み関数を使用してデバッグするには、次のようなルールに従う必要があります。

- ・ std_logic の読み込み操作では、ファイルに 0 と 1 以外の値 (X、Z など) があると、XST でデザインを合成できません。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。
- ・ 次の例にあるように、プロシージャを read で読み出す場合に条件文を使用すると、シミュレーションでエラーが発生します。

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

- ・ 次のようなスタイルで記述する場合に endfile 関数を使用したとします。

```
while (not endfile (MY_FILE)) loop
    readline (MY_FILE, MY_LINE);
    read (MY_LINE, MY_DATA);
end loop;
```

この場合、XST でデザインは合成できず、次のようなエラー メッセージが表示されます。

Line <MY_LINE> has not enough elements for target <MY_DATA>.

この問題を回避するには、次のように exit when endfile (MY_FILE); を while ループ内に追加します。

```
while (not endfile (MY_FILE)) loop
    readline (MY_FILE, MY_LINE);
    exit when endfile (MY_FILE);
    read (MY_LINE, MY_DATA);
end loop;
```

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。

- ・ 使用できる VHDL のデータ型
- ・ オーバーロード データ型
- ・ VHDL の多次元配列型

使用できる VHDL のデータ型

XST では、次の VHDL データ型を使用できます。

- ・ 列挙型
- ・ ユーザー定義の列挙型
- ・ ビット ベクタ型
- ・ 整数型
- ・ 定義済み型
- ・ STD_LOGIC_1164 IEEE 型

列挙型

データ型	値	意味	コメント
1 ビットの 2 進数	0, 1	--	--
BOOLEAN	false、true	--	--
REAL	\$-. から \$+.	--	--
STD_LOGIC	U	初期化されていない	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	L と同じように処理されます
	1	High	H と同じように処理されます
	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

ユーザー定義の列挙型

type COLOR is (RED, GREEN, YELLOW) ;

ビット ベクタ型

- ・ BIT_VECTOR
- ・ STD_LOGIC_VECTOR

制約が設定されていないもの（長さが指定されていないもの）は使用できません。

整数型

INTEGER

定義済み型

- ・ 1 ビットの 2 進数
- ・ BOOLEAN
- ・ BIT_VECTOR
- ・ INTEGER
- ・ REAL

STD_LOGIC_1164 IEEE 型

STD_LOGIC_1164 IEEE パッケージでは、次のデータ型が定義されています。

- ・ STD_LOGIC
- ・ STD_LOGIC_VECTOR

STD_LOGIC_1164 パッケージは IEEE ライブラリに含まれています。これらのデータ型を使用するには、次のように宣言する必要があります。

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;
```

オーバーロード データ型

次のデータ型はオーバーロード可能です。

- ・ オーバーロード 列挙型
- ・ オーバーロード ビット ベクタ型
- ・ オーバーロード 整数型
- ・ オーバーロード STD_LOGIC_1164 IEEE 型
- ・ オーバーロード STD_LOGIC_ARITH IEEE 型

オーバーロード 列挙型

- ・ STD_ULOGIC
STD_LOGIC データ型として 9 種類の論理値を持ちますが、定義済みの resolution 関数は含まれません。
- ・ X01
X、0、1 の値を含む STD_ULOGIC のサブタイプです。
- ・ X01Z
X、0、1、Z の値を含む STD_ULOGIC のサブタイプです。
- ・ UX01
U、X、0、1 の値を含む STD_ULOGIC のサブタイプです。
- ・ UX01Z
U、X、0、Z の値を含む STD_ULOGIC のサブタイプです。

オーバーロード ビット ベクタ型

- ・ STD_ULOGIC_VECTOR
- ・ UNSIGNED
- ・ SIGNED

制約が設定されていないもの（長さが指定されていないもの）は使用できません。

オーバーロード 整数型

- ・ NATURAL
- ・ POSITIVE

ユーザー定義範囲内の整数。たとえば、「type MSB is range 8 to 15;」は、7 より大きく 16 より小さい整数を表します。NATURAL および POSITIVE は、定義済みの VHDL データ型です。

オーバーロード STD_LOGIC_1164 IEEE 型

STD_LOGIC_1164 IEEE パッケージでは、次のデータ型が定義されています。

- ・ STD_ULOGIC (およびサブタイプ X01、X01Z、UX01、UX01Z)
- ・ STD_LOGIC
- ・ STD_ULOGIC_VECTOR
- ・ STD_LOGIC_VECTOR

このパッケージは、IEEE ライブラリにコンパイルされています。これらのデータ型を使用するには、次のように宣言する必要があります。

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;
```

オーバーロード STD_LOGIC_ARITH IEEE 型

UNSIGNED および SIGNED (STD_LOGIC の配列として定義) は、STD_LOGIC_ARITH IEEE パッケージで宣言されています。

このパッケージは、IEEE ライブラリにコンパイルされています。これらのデータ型を使用するには、次の 2 行を VHDL に追加する必要があります。

```
library IEEE;  
use IEEE.STD_LOGIC_ARITH.all;
```

VHDL の多次元配列型

XST では、3 次元までの多次元配列型がサポートされます。BRAM は推論されません。配列は、次のいずれかになります。

- ・ 信号
- ・ 定数
- ・ VHDL 変数

配列を使用すると、代入や数値演算を行うことができます。多次元配列を関数に渡し、インスタンスエーションで使用することもできます。

配列には、すべての次元に制約を指定する必要があります。次がその例です。

```
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;
```

また、次のように配列をマトリックスとして宣言できます。

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

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。

多次元配列の VHDL コード例 1

次のように宣言したとします。

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);
type TAB05 is array (4 downto 0) of WORD8;
type TAB03 is array (2 downto 0) of TAB05;

signal WORD_A : WORD8;
signal TAB_A, TAB_B : TAB05;
signal TAB_C, TAB_D : TAB03;
constant CNST_A : TAB03 := (
  ("00000000", "01000001", "01000010", "10000011", "00001100"),
  ("00100000", "00100001", "00101010", "10100011", "00101100"),
  ("01000010", "01000010", "01000100", "01000111", "01000100"));
```

この場合、次のように指定できます。

- ・ 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)**

多次元配列の VHDL コード例 2

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;
```

この場合、次のように指定できます。

- ・ 多次元配列の信号または変数 **MATRIXA <= CNST_A;**
- ・ An index of one row of the array: **MATRIXA (5) <= TAB_A;**
- ・ 最大の次元数のインデックス **MATRIXA (5,0) (0) <= '1';**

インデックスは、変数である可能性があります。

VHDL のレコード型

XST では、次のコード例に示すようにレコード型がサポートされます。

```
type REC1 is record
  field1: std_logic;
  field2: std_logic_vector (3 downto 0)
end record;
```

- ・ レコード型には、別のレコード型を含めることができます。
- ・ 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。
- ・ レコード信号に対して集合代入文がサポートされます。

VHDL の初期値

VHDL では、レジスタを宣言した際に初期値を設定できます。

初期値は、次の規則に従って設定する必要があります。

- ・ 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を使用できます。

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。

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

また、ビヘイビア記述の VHDL コードを使用して、レジスタにセット/リセットの値を指定できます。レジスタのリセットラインが最適な値になったときにレジスタに値を代入するには、次の例のように記述します。

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

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。

VHDL のローカルリセット/グローバルリセット

ローカルリセットは、グローバルリセットとは関係なく動作します。ローカルリセットで制御できるレジスタでは、グローバルリセット時（または電源投入時）とローカルリセット時で異なる値を指定できます。次のコード例では、レジスタ arb_onebit はグローバルリセット時には 1、ローカルリセット時 (rst) には 0 になるよう設定しています。

ローカルリセット/グローバルリセットの VHDL コード例

電源投入時にはレジスタの出力が 1 に初期化されるよう設定されていますが、この値はローカルリセットの値によって変わるので、ローカルセット/リセットがアクティブになると 0 に初期化されます。

```
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';

begin
  process (clk, rst)
  begin
    if rst='1' then
      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の概要

ザイリンクス FPGA デバイスに含まれるすべてのメモリエレメントは、既知のステートになる必要があるので、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、「ローカルリセット/グローバルリセットの VHDL コード例」の 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。

可能な場合、XST は信号の値を初期化する際に 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。

データ型	IEEE	XST
ビット	'0'	'0'
std_logic	'U'	'0'
bit_vector (3 downto 0)	0000	0000
std_logic_vector (3 downto 0)	0000	0000
integer (unconstrained)	integer'left	integer'left
integer range 7 downto 0	integer'left = 7	integer'left = 7 (coded as 111)
integer range 0 to 7	integer'left = 0	integer'left = 0 (coded as 000)
Boolean	FALSE	FALSE (coded as 0)
enum(S0,S1,S2,S3)	type'left = S0	type'left = S0 (coded as 000)

未接続の出力ポートは、デフォルトで VHDL 初期値の XST 列に示されている値になります。出力ポートに初期状態がある場合は、未接続の出力ポートが定義された初期状態になるように接続されます。IEEE 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必ずエラー 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 のオブジェクト

VHDL のオブジェクトには、次があります。

- ・ 信号
- ・ 変数
- ・ 定数

VHDL の信号

信号はアーキテクチャ宣言で宣言し、アーキテクチャ文内で使用します。また、信号をブロックで宣言し、そのブロック内で使用することも可能です。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

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

定数は宣言部で宣言でき、その部分で使用できます。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

サポートされる演算子は、「[VHDL の演算子](#)」にリストされています。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。

VHDL 演算子のコード例 1

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

これは、次と同じです。

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

VHDL 演算子のコード例 2

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

これは、次と同じです。

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

VHDL 演算子のコード例 3

```
sla (Shift Left Arithmetic)
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);
```

VHDL 演算子のコード例 4

```
sra (Shift Right Arithmetic)
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);
```

VHDL 演算子のコード例 5

```
rol (Rotate Left)
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);
```

VHDL 演算子のコード例 6

```
ror (Rotate Right)
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

VHDL のエンティティとアーキテクチャの記述には、次が含まれます。

- ・ 回路記述
- ・ エンティティ宣言
- ・ アーキテクチャ宣言
- ・ コンポーネントのインスタンス化
- ・ 再帰的コンポーネント インスタンス化
- ・ コンポーネント コンフィギュレーション
- ・ ジェネリック パラメータ宣言
- ・ ジェネリックと属性の競合

VHDL の回路記述

VHDL の回路記述は、次の 2 つの部分で構成されています。

- ・ インターフェイス (I/O ポートの定義)
- ・ 本体部分

VHDL では、それぞれ次に相当します。

- ・ エンティティはインターフェイス
- ・ アーキテクチャはビヘイビア

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、次が指定されます。

- ・ 名前
- ・ モード (in、out、inout、buffer)
- ・ タイプ (「[エンティティとアーキテクチャ宣言のコード例](#)」のポート A、B、C、D、E)

ポートタイプには制約を設定する必要があります。ポートには、1 次元配列型のみを使用できます。

VHDL のアーキテクチャ宣言

内部信号はアーキテクチャ内で宣言できます。各内部信号には、次が指定されます。

- ・ 名前
- ・ タイプ (「[エンティティとアーキテクチャ宣言のコード例](#)」の信号 T)

エンティティとアーキテクチャ宣言のコード例

```
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

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
コンポーネントは基本ブロックを表します。
- ・ ポート
ポートはコンポーネントの I/O コネクタを表します。
- ・ 信号
信号はコンポーネント間のワイヤに対応します。

VHDL の場合、コンポーネントはデザイン エンティティによって表されます。エンティティは、次から構成されます。

- ・ エンティティ宣言
エンティティ宣言では、コンポーネント ポートなどのコンポーネントを外側から見た「外観」が記述されます。
- ・ アーキテクチャ本体
アーキテクチャ本体では、ビヘイビアや構造などコンポーネントの「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 インスタンス化文で定義されます。この文では、別のコンポーネントのアーキテクチャ文内で使用されるコンポーネントのインスタンスを指定します。コンポーネント インスタンス化文はそれぞれ識別子で区別されます。

コンポーネント インスタンス化文では、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 (予約語ポート マップの後のかっこで囲まれたリスト) が含まれます。この関係リストでは、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。

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。

半加算器の構造記述のコード例

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;

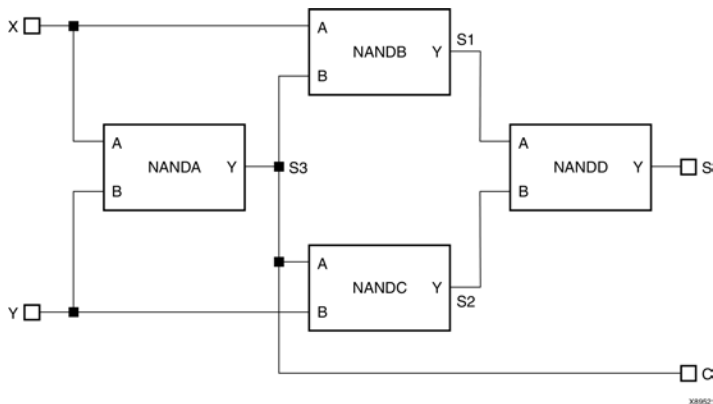
architecture ARCH1 of NAND2 is
begin
  Y <= A nand B;
end ARCH1;

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;
```

合成済みの最上位レベルのネットリストの図



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。ただし、直接的なインスタンスレーションは再帰的な文では使用できません。再帰呼び出しが永久に実行されるのを防ぐには、繰り返す回数をデフォルトの 4 で制限します。制限するには、-loop_iteration_limit オプションを使用します。

再帰的なコンポーネント インスタンスーション文を使用した 4 ビット シフト レジスタのコード例

```
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

エンティティ/アーキテクチャのペアをコンポーネント インスタンス文に関連付けると、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。

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

「半加算器の構造記述のコード例」は、コンポーネント インスタンスーション文でのコンフィギュレーション節の使用例を示しています。この例には、次の for all 文が含まれます。

```
for all : NAND2 use entity work.NAND2(ARCHI);
```

この文は、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 が使用されることを表します。

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 パラメータ宣言

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 パラメータの使用例としては、デザインのビット幅の設定があります。VHDL の場合、ジェネリック ポートで回路を記述すると、同じコンポーネントを異なるジェネリック ポートの値で繰り返しインスタンス化できるという利点があります (次のコード例を参照)。

ジェネリック ポートを使用した回路のコード例

```
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;
```

-generics コマンドライン オプションを使用すると、最上位レベルのデザイン ブロックで定義されるジェネリック (VHDL) 値を再定義できます。これにより、IP コアの生成やテストフローなどの HDL ソース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。詳細については、「[ジェネリック \(-generics\)](#)」を参照してください。

VHDL のジェネリックと属性の競合

ジェネリックおよび属性は、VHDL 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、また属性は制約ファイルでも指定できるので、競合が発生する場合があります。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) に指定されるものがコンポーネント (上位レベル) に指定されるものより優先されます。
2. ジェネリックと属性が同じインスタンスまたはコンポーネントに指定される場合、ジェネリックが優先され、XST で競合を示す警告メッセージが表示されます。
3. XST Constraint File (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。

インスタンスに指定されている属性がコンポーネントに指定されているジェネリックを上書きすると、シミュレーション ツールでそのジェネリックが使用されなくなる可能性があります。この結果、シミュレーションの結果が合成結果に一致しない場合があります。

VHDL での優先順位は、次の表に示す順になります。

VHDL での優先順位

	インスタンスのジェネリック	コンポーネントのジェネリック
インスタンスの属性	ジェネリックを適用 (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)
コンポーネントの属性	ジェネリックを適用	ジェネリックを適用 (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

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。

- ・ 同時処理信号代入文
- ・ generate 文
- ・ 組み合わせプロセス
- ・ if - else 文
- ・ case 文
- ・ for - loop 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。組み合わせロジックは、アーキテクチャ本体で定義します。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
- ・ 選択信号代入文
- ・ 条件的な信号代入文

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、アーキテクチャ部分で定義した同時処理信号の順序とは関係ありません。

同時処理信号代入文には次の 2 側面があります。

- ・ 左側
- ・ 右側

右側にある信号に変化 (イベント) が発生して代入が変わると、その計算結果が左側に代入されます。

シンプルな信号代入文の VHDL コード例

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

選択信号代入文を使用したマルチプレクサの VHDL コード例

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

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

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

条件的な信号代入文を使用したマルチプレクサの VHDL のコード例

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

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

VHDL のジェネレート文

反復構造は、VHDL の generate 文を使用して宣言します。たとえば、「for I in 1 to N generate」は、ビット スライス記述が N 回繰り返されることを意味します

for – generate 文を使用した 8 ビット加算器のコード例

次に、for – generate 文を使用して、ビットスライス構造の宣言により 8 ビット加算器を記述する例を示します。

```
entity EXAMPLE is
  port (
    A,B : in BIT_VECTOR (0 to 7);
    CIN : in BIT;
    SUM : out BIT_VECTOR (0 to 7);
    COUT : out BIT );
end EXAMPLE;

architecture ARCH1 of EXAMPLE 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 ARCH1;
```

スタティック条件でも if – generate 文がサポートされます。具体例は、「if – generate 文と for – generate 文を使用した N ビット加算器のコード例」を参照してください。この記述では、if – generate 文と for – generate 文を使用して、一般的な N ビット加算器 (ビット幅 4 ～ 32) を定義しています。

if – generate 文と for – generate 文を使用した N ビット加算器のコード例

```
entity EXAMPLE 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 EXAMPLE;

architecture ARCH1 of EXAMPLE is
  signal C : BIT_VECTOR (N+1 downto 0);
begin
  L1: 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 ARCH1;
```

VHDL の組み合わせプロセス文

プロセス文では、同時処理信号代入文とは異なる方法で信号に値が代入されます。値の代入は順次処理されます。ある代入によりその前の代入が無効になる場合があります。「プロセス文内で代入をしたコード例」を参照してください。まず信号 S に 0 が代入されますが、その後 (for (A and B) =1) で S の値が 1 に変化します。

推論されたハードウェアにメモリ エLEMENT が含まれない場合、プロセスは組み合わせプロセスとなります。つまり、プロセス文で指定されたすべての信号がプロセス文のすべての条件で代入される場合、プロセスは組み合わせプロセスとなります。

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

センシティブティリストに含まれていない信号がある場合、それを示す警告メッセージが表示され、センシティブティリストにその信号が追加されます。このため、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。

プロセスには、ローカル変数を含めることができます。変数は信号と同様に処理されますが、デザインには出力されません。

「組み合わせプロセスのコード例 1」では、プロセスの宣言部分で変数 AUX が宣言され、プロセス文で「:=」を使用して値が代入されています。

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。ラッチが作成されないようにするには、組み合わせプロセス文で定義したすべての信号がプロセスのすべての条件に対して常に明示的に代入されるようにします。

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 文
- ・ case 文
- ・ for - loop 文
- ・ 関数およびプロシージャ呼び出し

次のでは、各文の例を示します。

プロセス文内で代入をしたコード例

```
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;
```

組み合わせプロセスのコード例 1

```
library ASYL;
use ASYL.ARITH.all;

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 ARCHI 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 ARCHI;
```

組み合わせプロセスのコード例 2

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

architecture ARCHI of EXAMPLE is
  begin
    process (A,B)
      variable X, Y : BIT;
    begin
      X := A and B;
      Y := B and A;
      if X = Y then
        S <= '1' ;
      end if;
    end process;
  end ARCHI;
```

if – else 文

if – else 文では、真偽条件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

```
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、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。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
 - <
 - <=
 - >
 - >=
- ・ 次のいずれかに適合する次ステップの計算
 - `var = var + step`
 - `var = var - step`
 (var はループ変数、step は定数値)
- ・ next 文および exit 文はサポートされます。

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 mux4 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 の順序回路

順序回路は、順次プロセス文を使用して記述できます。XST では、次の 2 種類の文を使用できます。

- ・ 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 のセンシティビティ リスト付き順次プロセス文

プロセスが組み合わせプロセスでない場合、順次プロセスになります。つまり、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。

「非同期部分と同期部分のある順次プロセスのコード例」には、順序回路を記述したテンプレートになっています。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「[HDL コーディング手法](#)」を参照してください。

非同期部分と同期部分のある順次プロセスのコード例

非同期信号は、センシティビティリストで宣言する必要があります。宣言しない場合は警告メッセージが表示され、センシティビティリストに自動的に追加されます。この場合、合成結果のビヘイビアは最初の仕様と異なることがあります。

```
process (CLK, RST) ...
begin
  if RST = <'0' | '1'> then
    -- an asynchronous part may appear here
    -- optional part
    .....
  elsif <CLK'EVENT | not CLK'STABLE>
    and CLK = <'0' | '1'> then
    -- synchronous part
    -- sequential statements may appear here
  end if;
end process;
```

VHDL のセンシティビティ リストのない順次プロセス文

センシティビティリストのない順次プロセス文には、wait 文を含める必要があります。wait 文は、プロセス文の冒頭に記述します。wait 文の条件には、クロック信号の条件を使用します。プロセスに複数の wait 文を記述する場合は、一定の条件に従う必要があります。詳細は、「[VHDL での複数の wait 文の記述](#)」を参照してください。センシティビティリストのないプロセス文では、非同期信号イベントは指定できません。

センシティビティ リストなしの順次プロセスを記述したコード例

次の VHDL コード例では、で説明されているプロセス文の記述例を示します。クロック条件は立ち下りエッジまたは立ち上がりエッジです。

```
process ...
begin
  wait until <CLK'EVENT | not CLK' STABLE> and CLK = <'0' | '1'>;
  ... -- a synchronous part may be specified here.
end process;
```

XST では同じ wait 文 内でのクロックおよびクロック イネーブルの記述がサポートされないので、「クロックおよびクロック イネーブル (サポートなし) のコード例」に示すように記述します。

XST では、ラッチの記述に wait 文はサポートされていません。

クロックおよびクロック イネーブル (サポートなし) のコード例

```
wait until CLOCK'event and CLOCK = '0' and ENABLE = '1' ;
```

クロックおよびクロック イネーブル (サポートあり) のコード例

```
"8 Bit Counter Description Using a Process with a Sensitivity List" if ENABLE = '1' then ...
```

レジスタおよびカウンタの 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。アップデートは、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

センシティブリティ リスト付きプロセス文で記述した 8 ビット レジスタのコード例

```
entity EXAMPLE is
  port (
    DI  : in BIT_VECTOR (7 downto 0);
    CLK : in BIT;
    DO  : out BIT_VECTOR (7 downto 0) );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
  process (CLK)
  begin
    if CLK'EVENT and CLK = '1' then
      DO <= DI ;
    end if;
  end process;
end ARCHI;
```

センシティブリティ リストのないプロセス文 (wait 文あり) で記述した 8 ビット レジスタのコード例

```
entity EXAMPLE is
  port (
    DI  : in BIT_VECTOR (7 downto 0);
    CLK : in BIT;
    DO  : out BIT_VECTOR (7 downto 0) );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
  process begin
    wait until CLK'EVENT and CLK = '1';
    DO <= DI;
  end process;
end ARCHI;
```

クロック信号および非同期リセット信号を持つ 8 ビット レジスタのコード例

```
entity EXAMPLE is
  port (
    DI  : in BIT_VECTOR (7 downto 0);
    CLK : in BIT;
    RST : in BIT;
    DO  : out BIT_VECTOR (7 downto 0));
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
  process (CLK, RST)
  begin
    if RST = '1' then
      DO <= "00000000";
    elsif CLK'EVENT and CLK = '1' then
      DO <= DI ;
    end if;
  end process;
end ARCHI;
```

センシティビティ リスト付きプロセス文で記述した 8 ビット カウンタのコード例

```
library ASYL;
use ASYL.PKG_ARITH.all;

entity EXAMPLE is
  port (
    CLK : in BIT;
    RST : in BIT;
    DO  : out BIT_VECTOR (7 downto 0));
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
  process (CLK, RST)
    variable COUNT : BIT_VECTOR (7 downto 0);
  begin
    if RST = '1' then
      COUNT := "00000000";
    elsif CLK'EVENT and CLK = '1' then
      COUNT := COUNT + "00000001";
    end if;
    DO <= COUNT;
  end process;
end ARCHI;
```

VHDL での複数の wait 文の記述

順序回路は、プロセスで複数の wait 文を使用して記述できます。複数の wait 文を使用するには、次の規則に従う必要があります。

- ・ プロセスに loop 文を 1 つだけ含める
- ・ loop 文の冒頭に wait 文を記述する
- ・ wait 文の後に next 文または exit 文を使用する
- ・ すべての wait 文で同じ条件を使用する
- ・ wait 文の条件には 1 つのクロック信号のみを使用する
- ・ wait 文の条件には次のような形式を使用する

```
"wait [on clock_signal] until [(clock_signal'EVENT |      not clock_signal'STABLE)
and ] clock_signal = {'0' | '1'};"
```

複数の wait 文を使用した順次回路のコード例

次の VHDL コード例では、複数の wait 文を使用し、4 つの動作を連続して実行する順次回路の記述を示します。デザイン サイクルは、クロック信号の連続した 2 つの立ち上がりエッジで区切られています。また、動作シーケンスを最初から再開できるように、同期リセットが定義されています。この動作シーケンスでは、次の 4 入力をそれぞれ RESULT 出力に代入しています。

- ・ **DATA1**
- ・ **DATA2**
- ・ **DATA3**
- ・ **DATA4**

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity EXAMPLE is
  port (
    DATA1, DATA2, DATA3, DATA4 : in STD_LOGIC_VECTOR (3 downto 0);
    RESULT : out STD_LOGIC_VECTOR (3 downto 0);
    CLK : in STD_LOGIC;
    RST : in STD_LOGIC );
end EXAMPLE;

architecture ARCH of EXAMPLE is
begin
  process begin
    SEQ_LOOP : loop
      wait until CLK'EVENT and CLK = '1';
      exit SEQ_LOOP when RST = '1';
      RESULT <= DATA1;

      wait until CLK'EVENT and CLK = '1';
      exit SEQ_LOOP when RST = '1';
      RESULT <= DATA2;

      wait until CLK'EVENT and CLK = '1';
      exit SEQ_LOOP when RST = '1';
      RESULT <= DATA3;

      wait until CLK'EVENT and CLK = '1';
      exit SEQ_LOOP when RST = '1';
      RESULT <= DATA4;
    end loop;
  end process;
end ARCH;
```

VHDL の関数とプロシージャ

関数 (ファンクション) 宣言およびプロシージャ宣言は、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。ヘッダ部には次が含まれます。

- ・ 関数と入力には入力パラメータ
- ・ プロシージャには出力と入力パラメータ

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。

分解 (resolution) 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。

関数宣言と関数呼び出しのコード例

次に、関数をパッケージで宣言する VHDL の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。「プロシージャ宣言とプロシージャ呼び出しのコード例」では、プロシージャを使用して同じ機能を記述した例を示しています。

```
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 ARCH1 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 ARCH1;
```

プロシージャ宣言とプロシージャ呼び出しのコード例

```
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! 関数を示します。

```
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、VHDL デザインのエラーを検出できます。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。アサート文により、1 つのシフトレジスタのサイズが Shift Register LUT (SRL) のサイズを超えていないかどうかチェックされます。

SRL のサイズは 16 ビットであり、シフトレジスタの最後の段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 つめの SRL_WIDTH は 13、2 つめの SRL_WIDTH は 18 です。

シフトレジスタを記述する SINGLE_SRL のコード例

```
library ieee;
use ieee.std_logic_1164.all;

entity SINGLE_SRL is
  generic (SRL_WIDTH : integer := 16);
  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 (clk'event and clk = '1') then
      shift_reg <= shift_reg (SRL_WIDTH-1 downto 1) & 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;
```

シフトレジスタを記述する SINGLE_SRL のエラー メッセージ

「シフトレジスタを記述する SINGLE_SRL のコード例」を実行すると、アサート文により次のエラー メッセージが表示されます。

```
...
=====
*                      HDL Analysis                      *
=====
Analyzing Entity <top> (Architecture <beh>).
Entity <top> analyzed. Unit <top> generated.

Analyzing generic Entity <single_srl> (Architecture <beh>).
  SRL_WIDTH = 13
Entity <single_srl> analyzed. Unit <single_srl> generated.

Analyzing generic Entity <single_srl> (Architecture <beh>).
  SRL_WIDTH = 18
ERROR:Xst - assert_1.vhd line 15: FAILURE: The size of Shift Register exceeds the size of a single SRL
...
```

パッケージ文を使用した VHDL モデルの定義

VHDL モデルは、パッケージ文を使用して定義します。パッケージ文には、次が含まれます。

- ・ タイプおよびサブタイプ宣言
- ・ 定数宣言
- ・ 関数およびプロシージャ宣言
- ・ コンポーネント宣言

パッケージ文を使用すると、定数値や関数定義などのデザインのパラメータや定数を変更できます。

パッケージは、次から構成されます。

- ・ 本体の宣言
- ・ パッケージ宣言

パッケージ本体には、パッケージ宣言で宣言された関数本体の記述が含まれます。

パッケージは、XST で完全サポートされています。パッケージを使用するには、VHDL デザインの最初に次の行を含める必要があります。

```
library lib_pack;
-- lib_pack is the name of the library specified
-- where the package has been compiled (work by default)
use lib_pack.pack_name.all;
-- pack_name is the name of the defined package.
```

また、XST では定義済みのパッケージも使用できます。このパッケージはコンパイル済みで、VHDL デザインに含めることができます。これらのパッケージは主に合成中に使用するもののですが、シミュレーションでも使用できます。

標準パッケージ文を使用した VHDL モデルの定義

標準パッケージには、次の基本のデータ型が含まれます。

- ・ **bit**
- ・ **bit_vector**
- ・ **integer**

標準パッケージはデフォルトで含まれています。

IEEE パッケージを使用した VHDL モデルの定義

XST では、次の IEEE パッケージがサポートされます。

- ・ **std_logic_1164**
次がサポートされます。
 - **std_logic**
 - **std_ulogic**
 - **std_logic_vector**
 - **std_ulogic_vector**
 これらのデータ型に基づいた変換関数もサポートされます。
- ・ **numeric_bit**
ビット型を基にした次のベクタ型がサポートされます。
 - 符号なしベクタ
 - 符号付きベクタ
 次もサポートされます。
 - これらのデータ型のオーバーロードされた数値演算子すべて
 - これらのデータ型の変換および拡張関数
- ・ **numeric_std**
std_logic を基にした次のベクタ型がサポートされます。
 - 符号なしベクタ
 - 符号付きベクタ
 このパッケージは std_logic_arith と同等です。
- ・ **math_real**
次がサポートされます。
 - 「VHDL の実数定数」に記述される実数定数
 - 「VHDL の実数定数」に記述される実数関数
 - 0.0 ～ 1.0 の連続した値を生成するプロシージャ uniform

VHDL の実数定数

定数	値	定数	値
math_e	e	math_log_of_2	ln2
math_1_over_e	1/e	math_log_of_10	ln10
math_pi		math_log2_of_e	log2e
math_2_pi		math_log10_of_e	log10e
math_1_over_pi		math_sqrt_2	
math_pi_over_2		math_1_oversqrt_2	
math_pi_over_3		math_sqrt_pi	
math_pi_over_4		math_deg_to_rad	
math_3_pi_over_2		math_rad_to_deg	

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)	

実数 (real) 型と math_real パッケージの関数およびプロシージャは、計算にのみ使用します。XST での合成には使用できません。

次にコマンド例を示します。

```
library ieee;
use IEEE.std_logic_signed.all;
signal a, b, c : std_logic_vector (5 downto 0);
c <= a + b;
-- this operator "+" is defined in package std_logic_signed.
-- Operands are converted to signed vectors, and function "+"
-- defined in package std_logic_arith is called with signed
-- operands.
```

Synopsys パッケージ文を使用した VHDL モデルの定義

IEEE ライブラリでは、次の Synopsys パッケージがサポートされます。

- **std_logic_arith**
ベクタ型 (符号なし、符号付き) と、オーバーロードされた数値演算子が含まれます。変換関数や拡張関数も定義します。
- **std_logic_unsigned**
std_ulogic_vector で数値演算子を使用し、符号なし演算を定義します。
- **std_logic_signed**
std_logic_vector に対する数値演算子を、符号付き演算子として定義します。
- **std_logic_misc**
次の 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。
 - **and_reduce**
 - **or_reduce**

サポートされる VHDL 構文

XST では、次の VHDL コンストラクトがサポートされます。

- デザイン エンティティとコンフィギュレーション
- 論理式
- 文 (ステートメント)

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
整数型のみサポート
- ポート
制約の付いていないポート以外はサポート
- エンティティ文
サポートなし
- ・ VHDL パッケージ
STANDARD
TIME はサポートなし
- ・ VHDL の物理型
 - TIME
無視
 - REAL
サポートあり (定数計算関数でのみ)
- ・ VHDL モード
リンケージ
サポートなし
- ・ VHDL の宣言
データ型
列挙型、正の定数範囲を持つタイプ、ビット ベクタ型、多次元配列型をサポート
- ・ 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 (「[デザイン制約](#)」を参照)
- ・ VHDL の詳細設定
 - 属性
一部の定義済み属性 (**HIGH, LOW, 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

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)。

VHDL の論理式

XST では、次の論理式がサポートされます。

- ・ VHDL の演算子
- ・ VHDL のオペランド

VHDL の演算子

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
/, 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 のオペランド

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

XST では、次以外の VHDL 文がすべてサポートされます。

- ・ VHDL の wait 文
- ・ VHDL のループ文
- ・ VHDL の同時処理文

VHDL の wait 文

wait 文	サポートの有無
Boolean_expression まで sensitivity_list を待機状態にします。詳細は、「 VHDL の順次回路 」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 文の場合、各文に同じセンシティビティリストとブール代数式を使用する必要あり。 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。
time_expression の間、待機状態になります。詳細は、「 VHDL の順次回路 」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 ステートメント	サポートあり (遅延は無視)

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。
loop ... end loop	複数の wait 文でのみサポート

VHDL の同時処理文

同時処理文	サポートの有無
同時処理信号代入文 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
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 言語のサポート

この章では、XST での Verilog 構文とメタ コメントについて説明します。

- ・ [Verilog ビヘイビア記述](#)
- ・ [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 の制限](#)
- ・ [Verilog の属性とメタ コメント](#)
- ・ [サポートされる Verilog 構文](#)
- ・ [サポートされるシステム タスクと関数](#)
- ・ [Verilog プリミティブ](#)
- ・ [Verilog の予約言語](#)
- ・ [Verilog-2001 のサポート](#)

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](#)
[デザイン制約](#)
- ・ [Verilog 属性の構文](#)
[Verilog-2001 の属性](#)
- ・ [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。](#)
[一般制約](#)

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 として 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) 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。

Verilog ビヘイビア記述

Verilog のビヘイビア記述については、「[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 には、変数を使用してベクタから部分的にビットを選択できる機能が追加されています。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代わりに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で選択されます。

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
- ・ 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 ポートなどの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 ボディでは、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には、ローカル コンポーネント宣言で宣言されたコンポーネントに名前を指定し、実際の信号やポートをコンポーネント宣言のどのローカル ポートと接続するかを指定する関係リスト (かっこで囲まれたリスト) を含めます。

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

基本的な XOR 構築コード例

次は、2 つの 1 ビット入力 a および b を持つ基本的な XOR 関数の記述例です。

```
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。

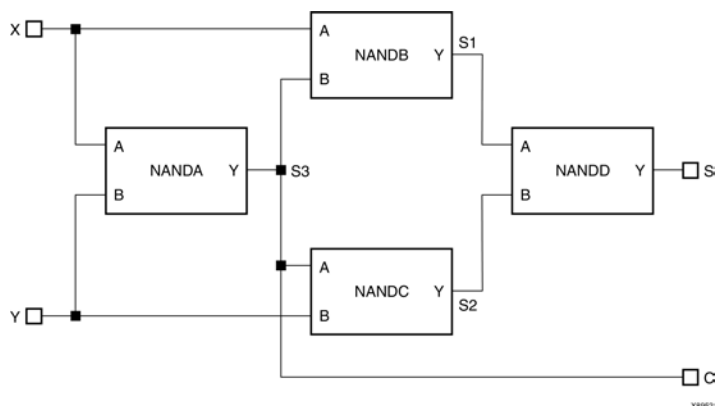
- ・ a_inv
- ・ b_inv
- ・ out

半加算器の構造記述例 (Verilog)

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
```

合成された最上位ネットリスト



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。

レジスタおよび BUFG の構造インスタンス化例

```
module foo (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。

```
(* BOX_TYPE="PRIMITIVE" *) // Verilog-2001
module FDC (Q, C, CLR, D);
parameter INIT = 1'b0;
output Q;
input C;
input CLR;
input D;
endmodule

(* BOX_TYPE="PRIMITIVE" *) // Verilog-2001
module BUFG ( O, I);
output O;
input I;
endmodule
```

Verilog パラメータ

Verilog モジュールでは、パラメータという定数を定義できます。パラメータは、任意のビット幅の回路を定義するためにモジュール インスタンスに渡されます。パラメータは、デザインに含まれるパラメータ指定したブロックを作成および使用する際の基礎となり、階層構造の作成やモジュール デザインで利用できます。

Verilog パラメータのコード例

次の Verilog コード例では、パラメータを使用しています。null 文字列パラメータは、サポートされていません。

```
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。

ジェネリック (-generics) コマンドライン オプションを使用すると、最上位デザイン ブロックで定義されるパラメータ (Verilog) 値を再定義できます。これにより、IP コアの生成やテストフローなどの HDL ソース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。

Verilog パラメータと属性の競合

パラメータおよび属性は、Verilog 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、また属性は制約ファイルでも指定できるので、競合が発生する場合があります。

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) 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. XST Constraint File (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。

インスタンスに指定されている属性が XST でモジュールに指定されているパラメータを上書きすると、シミュレーション ツールでそのパラメータが使用されなくなる可能性があります。この結果、シミュレーションの結果が合成結果と一致しない場合があります。

Verilog パラメータと属性の優先順位

優先順位は、次の表に示す順になります。

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 の制限

このセクションでは、XST での Verilog の制限について説明します。

Verilog の大文字/小文字の区別

Verilog では大文字と小文字が区別されるため、モジュールやインスタンス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が、ファイル名や混合言語、その他ツールとの互換性を考慮し、大文字や小文字の違いに頼らず、固有の名前にすることをお勧めします。

XST では、大文字/小文字が違うだけのモジュール名は使用できず、デザイン フローのその他の大文字/小文字の区別がないツールで問題がないように、インスタンスや信号名が変更されます。

大文字/小文字に関するXST のサポート

XST では、次のように大文字/小文字の区別がサポートされています。

- ・ I/O ポート、ネット、レジスタ、メモリには、大文字/小文字が異なる同じ名前を使用できます。
- ・ 同じ名前には、接尾辞が付けられます (rnm<Index>)。
- ・ 名前を変更する構文は NGC ファイルで生成されます。
- ・ 大文字/小文字のみが異なる Verilog 識別子を使用できます。これらの識別子には、XST により接尾辞が付けられます。

例：

```
module upperlower4 (input1, INPUT1, output1, output2);
  input input1;
  input INPUT1;
```

この例の場合、INPUT1 は INPUT1_rnm0 という名前に変更されます。

XST 内での Verilogの制限

次のように、ブロック、タスク、関数などに同じ名前は使用できません。

```
...
always @(clk)
begin: fir_main5
  reg [4:0] fir_main5_w1;
  reg [4:0] fir_main5_W1;
```

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。

```
ERROR:Xst:863 - "design.v", line 6: Name conflict (<fir_main5/fir_main5_w1> and
<fir_main5/fir_main5_W1>)
```

モジュール名に大文字/小文字が異なる同じ名前は使用できません。

```
module UPPERLOWER10 (...);
...
module upperlower10 (...);
...
```

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。

```
ERROR:Xst:909 - Module name conflict (UPPERLOWER10 and upperlower10)
```

Verilog のブロッキングおよびノンブロッキング代入文

XST では、次の例のように信号をブロッキング代入文およびノンブロッキング代入文の両方で指定できません。

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

変数がブロッキングとノンブロッキング代入文の両方で指定されると、XST で次のエラー メッセージが表示されます。

```
ERROR:Xst:880 - "design.v", line n: Cannot mix blocking and non-blocking assignments on signal <out1>.
```

ビットおよびスライスにブロッキング代入文およびノンブロッキング代入文の両方を使用することもできません。

次の例では、実際にはブロッキング代入とノンブロッキング代入が混合しているわけではありませんが、エラーが発生します。

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

エラーは、ビット レベルではなく、信号レベルで確認されます。

このエラーが複数ある場合は、最初の 1 つのみがレポートされます。

信号が指定されている行が複数ある場合は、エラー メッセージに示される行番号が正しくないこともあります (信号が代入されている箇所が複数行に渡っていることがあるため)。

Verilog の整数処理

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

case 文での整数処理

case 文でビットサイズ指定の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。

```
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。

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

Verilog の属性とメタ コメント

XST では、Verilog-2001 形式の属性と Verilog のメタ コメントがサポートされています。Verilog-2001 の属性はよく使用されてきているので、使用をお勧めします。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-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、モジュール宣言およびインスタンス化内、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します。
 - module
 - instance
 - net
- ・ 次の合成制約を設定してください。
 - フル ケース (FULL_CASE)
 - パラレル ケース (PARALLEL_CASE)

Verilog のメタ コメント

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します。
 - モジュール
 - インスタンス
 - ネット
- ・ 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

詳細は、「[デザイン制約](#)」を参照してください。

メタコメントの記述には、C 言語スタイル (`/* ... */`) または Verilog スタイル (`// ...`) を使用できます。C 言語スタイルでは、コメントを複数行にできます。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 \(TRANSLATE_ON\)](#) 制約

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

- ・ [パラレル ケース \(PARALLEL_CASE\)](#) 制約

```
// 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 のメタ コメントのコード例

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

サポートされる Verilog 構文

このセクションでは、次の XST でサポートされる Verilog 構文について説明します。

- ・ 定数
- ・ データ型
- ・ 継続代入文
- ・ 手続き代入文
- ・ デザイン階層
- ・ コンパイラ制約

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)。

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
文字列定数	サポートなし

サポートされる Verilog のデータ型

次の例外を除き、XST ではすべての Verilog データ タイプがサポートされています。

- ・ ネット タイプ
tri0、tri1、triereg はサポートされません。
- ・ 駆動電流
駆動電流はすべて無視されます。
- ・ レジスタ
real および realtime レジスタはサポートされません。
- ・ 名前付きイベント
名前付きイベントはすべてサポートされません。

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

サポートされる Verilog の手続き代入文

次の例外を除き、XST では Verilog の手続き代入文がサポートされています。

- ・ **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

・

サポートなし

・

サポートなし

・

無視

Disable

for および repeat ループ文以外はサポートあり

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

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

コンパイラ指示子	サポートの有無
<code>\uselib</code>	サポートなし
<code>\file, \line</code>	サポートあり

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。

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

`$signed(expr)` または `$unsigned(expr)`

これらの呼び出しの戻り値は入力値と同じサイズで、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。

`$readmemb` および `$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、「[外部ファイルからの RAM の初期化](#)」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 タスクでは次の例に示す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);
```

残りのシステム タスクは、処理中にコンピュータ画面およびログ ファイルに情報を出力するため、または合成中にファイルを開いて使用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 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
```

Verilog プリミティブ

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、次を除く Verilog のゲートレベルのプリミティブがサポートされます。

- ・ 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

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

always	and	assign	automatic
begin	buf	bufif0	bufif1
case	casex	casez	cell*
cmos	config*	deassign	default
defparam	design*	disable	edge
else	end	endcase	endconfig*
endfunction	endgenerate	endmodule	endprimitive
endspecify	endtable	endtask	event
for	force	forever	fork
function	generate	genvar	highz0
highz1	if	ifnone	incdir*
include*	initial	inout	input
instance*	integer	join	large
liblist*	library*	localparam*	macromodule
medium	module	nand	negedge
nmos	nor	noshow-cancelled*	not
notif0	notif1	or	output
parameter	pmos	posedge	primitive
pull0	pull1	pullup	pulldown
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	trireg	use*
vectored	wait	wand	weak0
weak1	while	wire	wor
xnor	xor		

Verilog-2001 のサポート

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

この章では、XST での Verilog ビヘイビア記述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 ビヘイビア記述のローカル リセット
- ・ Verilog ビヘイビア記述の配列
- ・ 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 ビヘイビア記述のモジュール宣言
- ・ 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、整数または実数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。

Verilog ビヘイビア記述の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;
```

この場合、それぞれ次を表します。

- ・ `arb_request[31]` は MSB
- ・ `arb_request[0]` は LSB

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。

初期値は、次の規則に従って設定する必要があります。

- ・ 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。

Verilog ビヘイビア記述の初期値のコード例

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。

```
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。

```
always @(posedge clk)  
begin  
    if (rst)  
        arb_onebit <= 1'b0;  
    end  
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 ビヘイビア記述のローカルリセット

ローカルリセットは、グローバルリセットとは関係なく動作します。ローカルリセットで制御できるレジスタでは、グローバルリセット時（または電源投入時）とローカルリセット時で異なる値を指定できます。次のコード例では、レジスタ `arb_onebit` はグローバルリセット時には 0、ローカルリセット時 (`rst`) には 1 になるよう設定しています。

Verilog ビヘイビア記述のローカル リセットのコード例

```
module mult(clk, rst, A_IN, B_OUT);
  input clk, rst, A_IN;
  output B_OUT;

  reg arb_onebit = 1'b0;

  always @(posedge clk or posedge rst)
  begin
    if (rst)
      arb_onebit <= 1'b1;
    else
      arb_onebit <= A_IN;
    end
  end
  B_OUT <= arb_onebit;
endmodule
```

電源投入時にはレジスタの出力が 0 に初期化されるよう設定されていますが、ローカル セット/リセットがアクティブになると 1 に初期化されます。

Verilog ビヘイビア記述の配列のコード例

コード例は、本書が作成された時点のものです。アップデートは、[ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip](http://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip) からダウンロードしてください。

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

次の例は、4 ビット幅のエLEMENT が 32 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。

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

Verilog 構造記述の配列のコード例

次の例は、8 ビット幅のエLEMENT 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エLEMENT 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。

多次元配列のコード例 1

次の例は、8 ビット幅の wire ELEMENT 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];
```

多次元配列のコード例 2

次の例は、64 ビット幅のレジスタ ELEMENT 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。

- ・ ネット
 - wire
 - tri
 - triand/wand
 - trior/wor
- ・ レジスタ
 - reg
 - integer
- ・ サプライ ネット
 - supply0
 - supply1
- ・ 定数
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を示します。

変数および信号の代入文

- ・ 変数 = 論理式
- ・ if (条件) 文
- ・ 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

論理式では、定数および変数に対し、「Verilog ビヘイビア記述でサポートされる演算子」の演算子を使用して、数値演算、論理演算、関係演算、条件演算が実行されます。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 ビヘイビア記述でサポートされる演算子

数値演算	論理演算	関係演算	条件的な信号代入文
+	&	<	?
-	&&	==	
*		===	
**		<=	
/	^	>=	
%	~	>=	
	~~	!=	
	~~	!==	
	<<	>	
	>>		
	<<<		
	>>>		

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
/	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	
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
べき乗	**	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
除算	/	サポートあり 符号付きの定数と符号なしの定数の間で除算を指定すると、不正なロジックが生成される 例 -1235/3'b111
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

論理式	シンボル	サポートの有無
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
条件的な信号代入文	?:	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

次の表に、XST で頻繁に使用される演算子を使用した論理式の評価結果を示します。

=== および != は、シミュレーションで変数に値 x または z が割り当てられているかを調べるのに便利な比較演算子です。合成中は、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。

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。コンポーネント間の接続は、モジュール インスタレーション文で定義されます。モジュール インスタンス文は、モジュールのインスタンスを指定します。モジュール インスタンス文には、インスタンス名が含まれます。また、実際のネットまたはポートを接続するモジュール宣言のローカル ポート (フォーマル) を指定する接続リストも含まれます。

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宣言

回路の I/O ポートはモジュール宣言文で宣言します。各ポートには、次が指定されます。

- ・ 名前
- ・ モード
 - **in**
 - **out**
 - **inout**

EXAMPLE というモジュールで宣言されている入力ポートおよび出力ポートは、デザインの基本的な入力および出力 I/O 信号です。Verilog の inout ポートはデバイス上の双方向 I/O ピンに対応し、データフローの方向はトライステートバッファへのイネーブル信号で制御されます。

次の例では、E はアクティブ High の出力イネーブル信号付きトライステートバッファとして記述されています。

- ・ `oe = 1` の場合は、信号 A の値は E の出力値になります。
- ・ `oe = 0` の場合はバッファはハイインピーダンス (z) になり、外部ロジックから E に入力された値はデバイスに取り込まれ、信号 D に送信されます。

Verilog ビヘイビア記述のモジュール宣言のコード例

```
module EXAMPLE (A, B, C, D, E);
  input A, B, C;
  output D;
  inout E;
  wire D, E;
  ...
  assign E = oe ? A : 1'bz;
  assign D = B & E;
  ...
endmodule
```

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 文の後に代入式を定義します。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。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。

継続代入文は、wire および tri データ型のみに使用可能です。

コード例は、本書が作成された時点のものです。アップデートは、http://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

assign 文を使用した継続代入文の例

```
wire par_eq_1;
....
assign par_eq_1 = select ? b : a;
```

assign 文を使用しない継続代入文の例

```
wire temp_hold = a | b;
```

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

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

Verilog ビヘイビア記述の組み合わせ 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。

- ・ #
- ・ @

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 @ の後にかつ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。ラッチが生成されないようにするには、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

Verilog ビヘイビア記述の if - else 文

if - else 文では、真偽条件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。

Verilog ビヘイビア記述の 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
```

Verilog ビヘイビア記述の 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、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。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、すべてのビット位置の z の値をドントケアとして認識します。

casex は、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。

Verilog ビヘイビア記述の 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、次の例に示すように 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。

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

Verilog ビヘイビア記述の 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
- ・ 演算子の <, <=, > または >= を使用したテスト条件の停止
- ・ 次のいずれかに適合する次ステップの計算

- `var = var + step`

- `var = var - step`

(var はループ変数、step は定数値)

repeat 文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。

for ループ文のコード例

```
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
```

Verilog ビヘイビア記述の While ループ

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 `disable <blockname>` なので、ラベルが付いているブロック内に使用する必要があります。

while ループ文のコード例

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

Verilog ビヘイビア記述の順次 always ブロック

順序回路は、センシティビティリストと共に always ブロックで記述します。センシティビティリストには、最大で次の 3 つのイベントが含まれます。

- ・ クロック信号イベント (必須)
- ・ リセット信号イベント (含まれる可能性あり)
- ・ セット信号イベント

こういった場合の always ブロックでは、if - else 文は 1 度しか使用できません。

非同期部分は、if - else 文の 1 番目および 2 番目の分岐で同期部分の前に記述できます。この場合、非同期部分で指定される信号には、次の定数値が代入されます。

- ・ 0
- ・ 1
- ・ x
- ・ z
- ・ これらを組み合わせたベクタ

これらの信号には、同期部分 (if...else 文の最後の分岐) でも値を代入されます。クロック信号の状態は、if...else 文の最後の分岐の状態になります。

always ブロックを使用した 8 ビット レジスタのコード例

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

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

always ブロックを使用した非同期リセット (アクティブ High) 付き 8 ビット レジスタのコード例

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

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

always ブロックを使用した非同期リセット付き 8 ビット カウンタのコード例

```
module seq2 (CLK, RST, DO);
  input CLK, RST;
  output [7:0] DO;
  reg [7:0] DO;

  always @(posedge CLK or posedge RST)
    if (RST == 1'b1)
      DO <= 8'b00000000;
    else
      DO <= DO + 8'b00000001;
endmodule
```

Verilog ビヘイビア記述の assign 文および deassign 文

assign 文および deassign 文は、単純なテンプレート内でサポートされています。

assign および deassign 文を使用した一般的なテンプレートのコード例

```
module assign (RST, SELECT, STATE, CLOCK, DATA_IN);
    input RST;
    input SELECT;
    input CLOCK;
    input [0:3] DATA_IN;
    output [0:3] STATE;
    reg [0:3] STATE;

    always @ (RST)
        if(RST)
            begin
                assign STATE = 4'b0;
            end
        else
            begin
                deassign STATE;
            end

    always @ (posedge CLOCK)
        begin
            STATE <= DATA_IN;
        end
endmodule
```

XST では、assign/deassign 文のサポートに次のような制限があります。

- ・ 1 つの信号に対しては、1 つの assign/deassign 文しか使用できません。
- ・ assign/deassign 文は、always ブロック内で if - else 文を使用して記述する必要があります。
- ・ assign/deassign 文で、信号の 1 つのビットまたは部分的ビット選択を指定することはできません。

assign/deassign 文のコード例

1 つの信号に対しては、1 つの assign/deassign 文しか使用できません。たとえば、次のようなデザインは XST ではサポートされません。

```
module dflop (RST, SET, STATE, CLOCK, DATA_IN);
    input RST;
    input SET;
    input CLOCK;
    input DATA_IN;
    output STATE;
    reg STATE;

    always @ (RST) // block b1
        if(RST)
            assign STATE = 1'b0;
        else
            deassign STATE;

    always @ (SET) // block b1
        if(SET)
            assign STATE = 1'b1;
        else
            deassign STATE;

    always @ (posedge CLOCK) // block b2
        begin
            STATE <= DATA_IN;
        end
endmodule
```

同じ always ブロックで実行される assign/deassign 文のコード例

assign/deassign 文は、always ブロック内で if - else 文を使用して記述する必要があります。たとえば、次のようなデザインは XST ではサポートされません。

```
module dflop (RST, SET, STATE, CLOCK, DATA_IN);
    input RST;
    input SET;
    input CLOCK;
    input DATA_IN;
    output STATE;

    reg STATE;

    always @ (RST or SET)           // block b1
    case ({RST,SET})
        2'b00: assign STATE = 1'b0;
        2'b01: assign STATE = 1'b0;
        2'b10: assign STATE = 1'b1;
        2'b11: deassign STATE;
    endcase

    always @ (posedge CLOCK)        // block b2
    begin
        STATE <= DATA_IN;
    end
endmodule
```

assign/deassign 文で信号のビット/部分的ビット選択ができない場合のコード例

assign/deassign 文で、信号の 1 つのビットまたは部分的ビット選択を指定することはできません。たとえば、次のようなデザインは XST ではサポートされません。

```
module assig (RST, SELECT, STATE, CLOCK, DATA_IN);
    input RST;
    input SELECT;
    input CLOCK;
    input [0:7] DATA_IN;
    output [0:7] STATE;

    reg [0:7] STATE;

    always @ (RST)                 // block b1
    if(RST)
    begin
        assign STATE[0:7] = 8'b0;
    end
    else
    begin
        deassign STATE[0:7];
    end

    always @ (posedge CLOCK)        // block b2
    begin
        if (SELECT)
            STATE [0:3] <= DATA_IN[0:3];
        else
            STATE [4:7] <= DATA_IN[4:7];
    end
end
```

Verilog ビヘイビア記述の 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。
 - 正の場合は 0 でパディングされます。
 - 負の場合は 1 でパディングされます。
 - ハイ インピーダンスの場合は z でパディングされます。
 - 不明の場合は x 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。

上記のルールは Verilog 2001 規格に準拠しており、Verilog 1995 とは互換性がありません。

Verilog ビヘイビア記述のタスクおよび関数

関数およびタスク宣言は、デザインでブロックを複数回使用する場合に有益です。関数およびタスクは、モジュール内で宣言して使用します。関数のヘッダには入力パラメータのみが、タスクのヘッダには入力、出力、入出力のパラメータが含まれます。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。

次のコード例では、モジュール内での関数宣言を示しています。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。

次のコード例にタスクを使用したコードは、この例の後に記述します。

モジュール内での関数宣言のコード例

```
module comb15 (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
```

タスク宣言とタスク イネーブルのコード例

次のコード例では、上記の例にタスクを使用しています。

```
module EXAMPLE (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のサポートが追加されています。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には、繰り返す回数をデフォルトの 4 で制限します。制限するには、-loop_iteration_limit オプションを使用します。

Verilog ビヘイビア記述の再帰を使用した構文のコード例

```
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

Verilog-2001 では、定数関数のサポートが追加されています。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。

定数関数を評価するコード例

```
module rams_cf (clk, we, a, di, do);
parameter DEPTH=1024;
input clk;
input we;
input [4:0] a;
input [3:0] di;
output [3:0] do;

reg [3:0] ram [size(DEPTH):0];

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

function integer size;
input depth;
integer i;
begin
size=1;
for (i=0; 2**i<depth; i=i+1)
size=i+1;
end
endfunction

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

タイミング制御文で # 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構文例

次に、ブロッキング手続き代入文の構文例を示します。

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

または

```
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われます。

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構文例

次は、ノンブロッキング手続き代入文の構文例です。

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

ノンブロッキング手続き代入文の使用例

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、次の例に示すようにマクロを定義できます。

```
'define TESTEQ1 4'b1101
```

定義されたマクロは、この後のデザイン コードで次のように参照します。

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

別の例を次に示します。

```
'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。

次に 'ifdef と 'endif の使用例を示します。

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

Verilog マクロ (-define) コマンドライン オプションを使用すると、Verilog マクロを定義または再定義できるので、これにより、IP コアの生成やテストベンチ フローなどの Hardware Description Language (HDL) ソース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。

Verilog ビヘイビア記述の include ファイル

Verilog では、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。

1 つの 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、複数の技術者が開発にかかわる場合に、デザインを複数のモジュールに分割してコードを記述できるため便利です。

‘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。

競合が発生する場合があるので注意してください。たとえば、Verilog ファイルの最初の部分に次があるとします。

```
`timescale 1 ns/1 ps
`include "modules.v"
...
```

‘include 文で指定したファイル (この場合は modules.v) を ISE Design Suite プロジェクト ディレクトリに追加すると、競合が発生する場合があります。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。

```
ERROR:Xst:1068 - fifo.v, line 2. Duplicate declarations of module 'RAMB4_S8_S8'
```

Verilog ビヘイビア記述のコメント

Verilog ビヘイビア記述では次の表に示す 2 つの形式でコメントを記述できます。Verilog ビヘイビア記述のコメント記述方法は、C++ などのプログラム言語と類似しています。

Verilog ビヘイビア記述のコメント型

シンボル	説明	使用目的	例
//	ダブル フォワード スラッシュ	1 行のコメント	// Define a one-line comment as illustrated by this sentence
/*	Slash asterisk	Multi-line comments	/* Define a multi-line comment by enclosing it as illustrated by this sentence */

Verilog ビヘイビア記述の generate 文

generate 文は、条件文からダイナミックに Verilog コードを作成するための構文です。この文を使用すると、繰り返し構造や、特定の条件の下でのみ適切な構造を作成できます。

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

XST では、次のタイプの generate 文がサポートされています。

- ・ Verilog ビヘイビア記述の generate – for 文
- ・ Verilog ビヘイビア記述の generate – if – else 文
- ・ Verilog ビヘイビア記述の generate – case 文

Verilog ビヘイビア記述の generate – for 文

generate – for ループ文は、モジュール内に 1 つ以上のインスタンスを作成します。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

generate – for ループ文を使用した 8 ビット加算器のコード例

```
generate
genvar i;

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

Verilog ビヘイビア記述の generate – if – else 文

generate – if – else 文は、条件を使用して生成するオブジェクトを制御する際に使用できます。

generate – if – else 文のコード例

次に、generate – if – else 文の使用例を示します。generate では、インスタンス化される乗算器のタイプが制御されます。

- ・ if – else 文の各分岐の内容は begin 文と end 文で囲む
- ・ begin 文には固有の修飾子が付いた名前を使用

```
generate
    if (IF_WIDTH < 10)
        begin : if_name
            adder # (IF_WIDTH) u1 (a, b, sum_if);
        end
    else
        begin : else_name
            subtractor # (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、generate ブロックの中で生成されるオブジェクトを条件によって制御する際に使用します。テストする条件が複数ある場合に、generate – case 文を使用して生成されるコードを決定します。

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

generate - case 文のコード例

次に、generate - case 文の使用例を示します。generate では、インスタンス化される加算器のタイプが制御されます。

```
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

この章では、Verilog/VHDL 混合の XST プロジェクトの実行方法について説明します。

- ・ [混合言語のプロジェクト ファイル](#)
- ・ [混合言語プロジェクトのVHDL/Verilog の境界規則](#)
- ・ [混合言語プロジェクトのポート マップ](#)
- ・ [混合言語プロジェクトのジェネリック サポート](#)
- ・ [混合言語プロジェクトのライブラリ検索順 \(LSO\) ファイル](#)

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。
 - VHDL デザインには Verilog モジュールをインスタンス化でき、
 - Verilog デザインには VHDL エンティティをインスタンス化できます。
 - それ以外の方法で VHDL と Verilog は混合不可能
- ・ 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、エラボーレーション段階で 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。
- ・ VHDL および Verilog プロジェクト ファイルは 1 つに統一されます。
- ・ VHDL および Verilog ライブラリが論理的に統一されます。
- ・ VHDL のみで可能だったコンパイル用の作業ディレクトリ (xsthdmdir) の指定が Verilog でも可能になります。
- ・ VHDL のみで可能だった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 のメカニズムが、Verilog でも可能になります。
- ・ 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。エラボー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。

混合言語のプロジェクト ファイル

XST では、VHDL/Verilog 混合デザインをサポートするため、専用の混合言語プロジェクトファイルを使用します。この混合言語フォーマットは、混合言語プロジェクトだけではなく、VHDL のみまたは Verilog のみのプロジェクトにも使用できます。

- ・ ISE® Design Suite から XST を実行する場合は、ISE でプロジェクト ファイルが作成されます。このプロジェクト ファイルは、常に混合言語です。
- ・ コマンドラインから XST を起動する場合は、混合言語プロジェクト用にユーザーがプロジェクト ファイルを作成する必要があります。

コマンドラインで混合言語プロジェクトファイルを作成するには、`-ifmt` コマンドライン オプションを `mixed` または値なしで使用します。既存のデザインに関しては、VHDL および Verilog 形式をそのまま使用できます。VHDL 形式を使用する場合は `-ifmt` を `vhdl` に設定し、Verilog 形式を使用する場合は `-ifmt` を `verilog` に設定します。

混合言語プロジェクトでライブラリなど外部ファイルを呼び出す場合は、次の構文を使用します。

```
language library file_name.ext
```

次に、混合言語プロジェクトでライブラリを呼び出す例を示します。

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

1 つの行で 1 つの Hardware Description Language (HDL) デザイン ファイルを指定します。

- ・ 最初の列は、HDL ファイルが VHDL であるか Verilog であるかを指定します。
- ・ 2 番目の列は、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。
- ・ 3 番目の列は、HDL ファイルの名前を指定します。

混合言語プロジェクトの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 デザインには Verilog モジュールをインスタンス化でき、Verilog デザインには VHDL エンティティをインスタンス化できます。

VHDL デザインへの Verilog モジュールのインスタンス化

VHDL デザインに Verilog モジュールをインスタンス化するには、次の手順に従います。

1. インスタンス化する 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。
 - a. ISE® Design Suite から [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] にある [Case] プロパティを [Maintain] に設定
 - b. コマンドラインで `-case` オプションを `maintain` に設定
2. VHDL コンポーネントをインスタンス化すると同様に、Verilog コンポーネントをインスタンス化します。

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、「[混合言語プロジェクトのライブラリ検索順 \(LSO\) 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。

ライブラリが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。

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

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

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

Verilog デザインにインスタンスエートできる VHDL の構文は、VHDL エンティティのみです。その他の VHDL の構文は Verilog コードで認識されません。XST では、エンティティ/アーキテクチャ ペアが Verilog と VHDL の境界として使用されます。

XST では、バインド処理はエラボレーション段階で行われます。バインド処理では、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、「[混合言語プロジェクトのライブラリ検索順 \(LSO\) 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、インスタンスエ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、「[混合言語プロジェクトのライブラリ検索順 \(LSO\) 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エートする場合、XST では次のような制限があります。

- ・ ポートの関連付けは明示的に行う必要があります。ポート マップでは、必ず正式な有効ポート名を指定してください。
- ・ パラメータは、値が変化しない場合でも、インスタンスエート時にすべて渡す必要があります。
- ・ パラメータを変更する場合は、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エーションを使用してください。

パラメータ変更の正しいコード例

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

パラメータ変更の正しいコード例

次のような記述は XST で正しく認識されません。

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


混合言語プロジェクトのポート マップ

混合言語プロジェクトのポート マップには、次が含まれます。

- ・ Verilog 内の VHDL ポート マップ
- ・ VHDL 内の Verilog ポート マップ
- ・ 混合言語内の VHDL ポート マップ
- ・ 混合言語の Verilog ポート マップ

Verilog 内の VHDL ポート マップ

Verilog デザインにインスタンス化された VHDL エンティティでは、次のポートタイプがサポートされます。

- ・ **in**
- ・ **out**
- ・ **inout**

XST では、VHDL の buffer と linkage ポートはサポートされません。

VHDL 内の Verilog ポート マップ

VHDL デザインにインスタンス化された Verilog モジュールでは、次のポートタイプがサポートされます。

- ・ 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。

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、同等のコンポーネント宣言を使用してください。デフォルトでは、Verilog ポート名はすべて小文字であると判断されます。

混合言語内の VHDL ポート マップ

XST では、混合言語デザインで次の VHDL データ型がサポートされます。

- ・ **bit**
- ・ **bit_vector**
- ・ **std_logic**
- ・ **std_ulogic**
- ・ **std_logic_vector**
- ・ **std_ulogic_vector**

混合言語の Verilog ポート マップ

XST では、混合言語デザインで次の Verilog データ型がサポートされます。

- ・ **wire**
- ・ **reg**

混合言語プロジェクトのジェネリック 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`

混合言語プロジェクトのライブラリ検索順 (LSO) ファイル

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。デフォルトでは、プロジェクト ファイルに現れる順序でファイルが検索されます。

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 でのライブラリ検索順 (LSO) ファイルの指定

ISE® Design Suite では、ライブラリ検索順 (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。ISE Design Suite では、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。

コマンドラインでのライブラリ検索順 (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、`-lso` コマンドライン オプションを使用してライブラリ検索順 (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。

ライブラリ検索順 (LSO) に関する規則

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

- ・ LSO ファイルが空の場合
- ・ `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

LSO ファイルが空の場合

ライブラリ検索順 (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

LSO ファイルに DEFAULT_SEARCH_ORDER キーワードのみが含まれる場合、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      vllib1  f1.vhd
verilog    rtfl1lib f1.v
vhdl      vllib2  f3.vhd
LSO file Created by ProjNav
```

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。

```
vllib1
rtfl1lib
vllib2
```

検索後の my_proj.lso の内容は、次のとおりです。

```
vllib1
rtfl1lib
vllib2
```

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

LSO ファイルに 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。

```
vhdl      vllib1  f1.vhd
verilog    rtfl1lib f1.v
vhdl      vllib2  f3.vhd
```

LSO ファイル my_proj.lso の内容は、次のとおりです。

```
rtfl1lib
vllib2
vllib1
DEFAULT_SEARCH_ORDER
```

XST では、次の検索順が使用されます。

```
vllib1
rtfl1lib
vllib2
```

検索後の my_proj.lso の内容は、次のとおりです。

```
rtfl1lib
vllib2
vllib1
DEFAULT_SEARCH_ORDER
```

ライブラリ リストのみの場合

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

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

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      vllib1  f1.vhd
verilog   rtfl1lib f1.v
vhd1      vllib2  f3.vhd
```

LSO ファイル my_proj.lso の内容は、次のとおりです。

```
rtfl1lib
vllib2
vllib1
```

XST では、次の検索順が使用されます。

```
rtfl1lib
vllib2
vllib1
```

検索後の my_proj.lso の内容は、次のとおりです。

```
rtfl1lib
vllib2
vllib1
```

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 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。

```
vhd1      vllib1  f1.vhd
verilog   rtfl1lib f1.v
vhd1      vllib2  f3.vhd
```

LSO ファイル my_proj.lso の内容は、次のとおりです。

```
personal_lib
rtfl1lib
vllib2
vllib1
```

XST では、次の検索順が使用されます。

```
rtfl1lib
vllib2
vllib1
```

検索後の my_proj.lso の内容は、次のとおりです。

```
rtfl1lib
vllib2
vllib1
```


XST ログ ファイル

この章では、XST のログ ファイルについて説明します。

- ・ [FPGA のログ ファイルの内容](#)
- ・ [ログ ファイルの容量削減方法](#)
- ・ [ログ ファイルのマクロ](#)
- ・ [ログ ファイルの例](#)

FPGA のログ ファイルの内容

XST で出力される FPGA ログ ファイルには、次の情報が含まれています。

- ・ 著作権情報
- ・ 目次
- ・ 合成オプションのサマリ
- ・ HDL のコンパイル
- ・ デザイン階層の解析ツール
- ・ HDL 解析
- ・ HDL 合成レポート
- ・ アドバンス HDL 合成レポート
- ・ 下位レベルの合成
- ・ パーティション レポート
- ・ 最終レポート

FPGA ログ ファイルの著作権情報

著作権情報には、次が含まれます。

- ・ ISE® Design Suite のバージョン番号
- ・ ザイリンクスの著作権の表示

FPGA のログ ファイルの目次

FPGA のログ ファイルの目次には、そのログ ファイルの主なセクションがリストされます。この目次を使用して、別のセクションへすばやくナビゲートできます。目次にある項目は、リンクされていません。テキスト エディタの検索機能を使用して、ナビゲートしてください。

FPGA ログ ファイルの合成オプションのサマリ

合成オプションのサマリには、次に関する情報が含まれます。

- ・ ソース パラメータ
- ・ ターゲット パラメータ
- ・ ソース オプション
- ・ ターゲット オプション
- ・ 一般オプション
- ・ その他のオプション

FPGA ログ ファイルの HDL コンパイル

HDL コンパイルの詳細は、「[FPGA ログ ファイルの HDL 解析](#)」を参照してください。

FPGA ログ ファイルのデザイン階層解析

詳細は、「[FPGA ログ ファイルの HDL 解析](#)」を参照してください。

FPGA ログ ファイルの HDL 解析

HDL のコンパイル、デザイン階層解析、および HDL 解析中、XST では次が実行されます。

- ・ VHDL/Verilog ファイルの解析
- ・ デザイン階層の認識
- ・ コンパイルされるライブラリの命名

このステップで、合成とシミュレーション結果の間に見られる不一致などの問題が検出されます。

FPGA ログ ファイルの Hardware Description Language (HDL) 合成レポート

XST では Hardware Description Language (HDL) 合成中に基本的なマクロが可能な限り認識され、ターゲット アーキテクチャに合ったインプリメンテーションが作成されます。この作業はブロックごとに行われ、このステップの最終段階で HDL 合成レポートが出力されます。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「[HDL のコーディング手法](#)」を参照してください。

FPGA ログ ファイルのアドバンス HDL 合成レポート

XST では、次のような高度なマクロ認識およびマクロ推論が実行されます。この場合、XST は次のように動作します。

- ・ ダイナミック シフトレジスタなどの認識
- ・ パイプライン乗算器のインプリメンテーション
- ・ ステート マシンのコード記述

この部分には、デザイン全体で認識されたマクロのサマリがタイプ別に記述されます。

FPGA ログ ファイルの下位レベルの合成

XST では、潜在的な削除情報（等価フリップフロップの削除など）や、レジスタの複製などがレポートされます。詳細については、「[FPGA 最適化レポート セクション](#)」を参照してください。

FPGA ログ ファイルのパーティション レポート

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。

FPGA ログ ファイルの最終レポート

最終レポート部分には、次の情報が含まれます。

- ・ 次の最終結果
 - RTL 最上位レベルの出力ファイル名 (例 stopwatch.ngr)
 - 最上位レベルの出力ファイル名 (例 : stopwatch)
 - 出力ファイル形式 (例 : NGC)
 - 最適化ゴール (例 : Speed)
 - 階層維持のための制約 (KEEP_HIERARCHY) の使用の有無 (例 : No)
- ・ セル使用率
BEL、クロック バッファ、I/O バッファなどのセル使用率レポート
- ・ デバイス使用率のサマリ
XST により予測されたスライス、フリップフロップ、IOB、BRAM などの数が示されます。このレポートは、MAP で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
XST により予測された各パーティションのスライス、フリップフロップ、IOB、BRAM などの数が示されます。このレポートは、MAP で生成されるレポートと類似しています。
- ・ タイミング レポート
XST では合成の最後に詳細なタイミング情報がレポートされます。タイミング レポートには、ネットリストの 4 つの使用可能なドメインに関する情報が表示されます。
 - レジスタからレジスタ
 - 入力からレジスタ
 - レジスタから出力パッド
 - 入力パッドから出力パッド

この例は、「[FPGA ログ ファイルの例](#)」のタイミング レポート セクションにあります。詳細については、「[FPGA 最適化レポート セクション](#)」を参照してください。
- ・ 暗号化されたモジュール
デザインに暗号化されたモジュールが含まれる場合、これらのモジュール情報は非表示になります。

ログ ファイルの容量削減方法

XST のログ ファイルの容量削減方法

- ・ メッセージ フィルタの使用
- ・ Quiet モードの使用
- ・ Silent モードの使用
- ・ 特定メッセージの非表示

メッセージ フィルタの使用

ISE® Design Suite から XST を実行する場合は、メッセージ フィルタ ツールを使用してログ ファイルから特定のメッセージのみを表示できます。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 モードの使用

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。このモードを設定するには、`-intstyle` コマンド ライン 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、画面にすべてのログが出力されます。Quiet モードを使用すると出力されなくなる情報は、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
- ・ 次に示す最終レポートのセクション
 - CPLD デバイスの最終結果ヘッダ
 - FPGA デバイスの最終結果セクション
 - 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

FPGA デバイスの場合、このオプションを使用しても次のセクションは表示されます。

- ・ 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 モードの使用

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ず、ログ ファイルにのみメッセージが記述されます。このモードを使用するには、`-intstyle` オプションを `silent` に設定します。

特定メッセージの非表示

XIL_XST_HIDEMESSAGES 環境変数を使用すると、XST により HDL 合成または下位レベルの合成段階で生成される特定のメッセージを非表示にできます。この環境変数は、次の表のいずれかの値に設定します。

XIL_XST_HIDEMESSAGES 環境変数の値

値	意味
none (デフォルト)	すべてのメッセージが表示されます。
hdl_level	VHDL/Verilog 合成および HDL 基本合成、アドバンス合成で表示されるメッセージを削減します。
low_level	下位レベル合成で表示されるメッセージを削減します。
hdl_and_low_levels	すべての段階のメッセージを削減します。

hdl_level と hdl_and_low_levels に設定した場合に削除されるメッセージ

XIL_XST_HIDEMESSAGES 環境変数を hdl_level または hdl_and_low_levels に設定すると、次のメッセージが表示されなくなります。

- WARNING:HDLCompilers:38 – design.v line 5 Macro 'my_macro' redefined
メモ : このメッセージは、Verilog コンパイラを使用した場合のみ表示されます。
- WARNING:Xst:916 – design.vhd line 5: Delay is ignored for synthesis.
- WARNING:Xst:766 – design.vhd line 5: Generating a Black Box for component comp.
- Instantiating component comp from Library lib.
- Set user-defined property "LOC = X1Y1" for instance inst in unit block.
- Set user-defined property "RLOC = X1Y1" for instance inst in unit block.
- Set user-defined property "INIT = 1" for instance inst in unit block.
- Register reg1 equivalent to reg2 has been removed.

low_level または hdl_and_low_levels に設定した場合に削除されるメッセージ

XIL_XST_HIDEMESSAGES 環境変数を low_level または hdl_and_low_levels に設定すると、次のメッセージが表示されなくなります。

- WARNING:Xst:382 – Register reg1 is equivalent to reg2. Register reg1 equivalent to reg2 has been removed.
- WARNING:Xst:1710 – FF/Latch reg (without init value) is constant in block block.
- WARNING:Xst 1293 – FF/Latch reg is constant in block block.
- WARNING:Xst:1291 – FF/Latch reg is unconnected in block block.
- WARNING:Xst:1426 – The value init of the FF/Latch reg hinders the constant cleaning in the block block. You could achieve better results by setting this init to value.

ログ ファイルのマクロ

XST ログ ファイルには、ブロックごとに VHDL または Verilog ソースから推論される、マクロ セットおよび関連する信号などの詳細な情報が含まれます。

マクロの推論は、次の 2 段階で行われます。

1. HDL 合成

XST では、加算器、減算器、レジスタなどのできるだけ多くの単純なマクロ ブロックが認識されます。

2. アドバンス HDL 合成

XST では、HDL 合成段階で認識されたマクロを改善したり (例 : 乗算器のパイプライン接続)、ダイナミック シフト レジスタなどの複雑なマクロを新たに作成することで、マクロがさらに詳細に処理されます。アドバンス HDL 合成 段階でレポートされるマクロ認識レポートは、HDL 合成段階でのレポートに従ってフォーマットされます。

XST では、認識されたマクロの全体的な統計が次の 2 段階で表示されます。

- ・ HDL 合成段階の後
- ・ アドバンス HDL 合成段階の後

XST では、最終レポートに保護されたマクロの統計は表示されなくなっています。

ログ ファイルの例

このセクションでは、次の XST のログ ファイル例を示します。

- ・ 認識されたマクロのログ ファイル例
- ・ 詳細なマクロ処理のログ ファイル例
- ・ FPGA ログ ファイルの例
- ・ CPLD ログ ファイルの例

認識されたマクロのログ ファイル例

次のログ ファイルは、ブロックごとに認識されたマクロと、この段階の後の全体的なマクロの統計を示しています。

```
=====
*                      HDL Synthesis                      *
=====
...
Synthesizing Unit <decode>.
  Related source file is "decode.vhd".
  Found 16x10-bit ROM for signal <one_hot>.
  Summary:
    inferred    1 ROM(s).
Unit <decode> synthesized.

Synthesizing Unit <statmach>.
  Related source file is "statmach.vhd".
  Found finite state machine <FSM_0> for signal <current_state>.
-----
| States           | 6 |
| Transitions      | 11 |
| Inputs           | 1 |
| Outputs          | 2 |
| Clock            | CLK (rising_edge) |
| Reset            | RESET (positive) |
| Reset type       | asynchronous |
| Reset State      | clear |
| Power Up State   | clear |
| Encoding         | automatic |
| Implementation   | LUT |
-----
  Summary:
    inferred    1 Finite State Machine(s).
Unit <statmach> synthesized.
...
=====
HDL Synthesis Report

Macro Statistics
# ROMs                      : 3
  16x10-bit ROM             : 1
  16x7-bit ROM              : 2
# Counters                  : 2
  4-bit up counter          : 2
=====
...
```

詳細なマクロ処理のログ ファイル例

次の XST の FPGA ログ ファイルは、アドバンス HDL 合成中にさらに詳しく処理されたマクロと、この段階の後の全体的なマクロの統計を示しています。

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

Analyzing FSM <FSM_0> for best encoding.
Optimizing FSM <MACHINE/current_state/FSM_0> on signal <current_state[1:3]> with gray encoding.
-----
State      | Encoding
-----
clear      | 000
zero       | 001
start      | 011
counting   | 010
stop       | 110
stopped    | 111
-----

=====
Advanced HDL Synthesis Report

Macro Statistics
# FSMs                      : 1
# ROMs                      : 3
  16x10-bit ROM             : 1
  16x7-bit ROM              : 2
# Counters                  : 2
  4-bit up counter         : 2
# Registers                  : 3
  Flip-Flops/Latches       : 3

=====
...
```

FPGA ログ ファイルの例

次は、FPGA 合成の XST ログ ファイルの例です。

```
Release 10.1 - xst K.31 (nt64)

Copyright (c) 1995-2008 Xilinx, Inc. All rights reserved.

TABLE OF CONTENTS

1) Synthesis Options Summary
2) HDL Compilation
3) Design Hierarchy Analysis
4) HDL Analysis
5) HDL Synthesis
  5.1) HDL Synthesis Report
6) Advanced HDL Synthesis
  6.1) Advanced HDL Synthesis Report
7) Low Level Synthesis
8) Partition Report
9) Final Report
```

9.1) Device utilization summary

9.2) Partition Resource Summary

9.3) TIMING REPORT

=====

* Synthesis Options Summary *

=====

---- Source Parameters

Input File Name : "stopwatch.prj"

Input Format : mixed

Ignore Synthesis Constraint File : NO

---- Target Parameters

Output File Name : "stopwatch"

Output Format : NGC

Target Device : xc4vlx15-12-sf363

---- Source Options

Top Module Name : stopwatch

Automatic FSM Extraction : YES

FSM Encoding Algorithm : Auto

Safe Implementation : No

FSM Style : lut

RAM Extraction : Yes

RAM Style : Auto

ROM Extraction : Yes

Mux Style : Auto

Decoder Extraction : YES

Priority Encoder Extraction : YES

Shift Register Extraction : YES

Logical Shifter Extraction : YES

XOR Collapsing : YES

ROM Style : Auto

Mux Extraction : YES

Resource Sharing : YES

Asynchronous To Synchronous : NO

Use DSP Block : auto

Automatic Register Balancing : No
---- Target Options
Add IO Buffers : YES
Global Maximum Fanout : 500
Add Generic Clock Buffer(BUFG) : 32
Number of Regional Clock Buffers : 16
Register Duplication : YES
Slice Packing : YES
Optimize Instantiated Primitives : NO
Use Clock Enable : Auto
Use Synchronous Set : Auto
Use Synchronous Reset : Auto
Pack IO Registers into IOBs : auto
Equivalent register Removal : YES
---- General Options
Optimization Goal : Speed
Optimization Effort : 1
Power Reduction : NO
Library Search Order : stopwatch.lso
Keep Hierarchy : NO
Netlist Hierarchy : as_optimized
RTL Output : Yes
Global Optimization : AllClockNets
Read Cores : YES
Write Timing Constraints : NO
Cross Clock Analysis : NO
Hierarchy Separator : /
Bus Delimiter : <>
Case Specifier : maintain
Slice Utilization Ratio : 100
BRAM Utilization Ratio : 100
DSP48 Utilization Ratio : 100
Verilog 2001 : YES
Auto BRAM Packing : NO

Slice Utilization Ratio Delta : 5

=====

* HDL Compilation *

=====

Compiling verilog file "smallcntr.v" in library work

Compiling verilog file "statmach.v" in library work

Module <smallcntr> compiled

Compiling verilog file "hex2led.v" in library work

Module <statmach> compiled

Compiling verilog file "decode.v" in library work

Module <hex2led> compiled

Compiling verilog file "cnt60.v" in library work

Module <decode> compiled

Compiling verilog file "stopwatch.v" in library work

Module <cnt60> compiled

Module <stopwatch> compiled

No errors in compilation

Analysis of file <"stopwatch.prj"> succeeded.

Compiling vhdl file "C:/xst/watchver/tenths.vhd" in Library work.

Entity <tenths> compiled.

Entity <tenths> (Architecture <tenths_a>) compiled.

Compiling vhdl file "C:/xst/watchver/dcm1.vhd" in Library work.

Entity <dcm1> compiled.

Entity <dcm1> (Architecture <BEHAVIORAL>) compiled.

=====

* Design Hierarchy Analysis *

=====

Analyzing hierarchy for module <stopwatch> in library <work>.

Analyzing hierarchy for entity <dcm1> in library <work> (architecture <BEHAVIORAL>).

Analyzing hierarchy for module <statmach> in library <work> with parameters.

clear = "000001"

counting = "001000"

start = "000100"

stop = "010000"

stopped = "100000"

zero = "000010"

Analyzing hierarchy for module <decode> in library <work>.

Analyzing hierarchy for module <cnt60> in library <work>.

Analyzing hierarchy for module <hex2led> in library <work>.

Analyzing hierarchy for module <smallcnt> in library <work>.

=====

* HDL Analysis *

=====

Analyzing top module <stopwatch>.

Module <stopwatch> is correct for synthesis.

Analyzing Entity <dcm1> in library <work> (Architecture <BEHAVIORAL>).

Set user-defined property "CAPACITANCE = DONT_CARE" for instance <CLKIN_IBUFG_INST> in unit <dcm1>.

Set user-defined property "IBUF_DELAY_VALUE = 0" for instance <CLKIN_IBUFG_INST> in unit <dcm1>.

Set user-defined property "IOSTANDARD = DEFAULT" for instance <CLKIN_IBUFG_INST> in unit <dcm1>.

Set user-defined property "CLKDV_DIVIDE = 2.0000000000000000" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "CLKFX_DIVIDE = 1" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "CLKFX_MULTIPLY = 4" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "CLKIN_DIVIDE_BY_2 = FALSE" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "CLKIN_PERIOD = 20.0000000000000000" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "CLKOUT_PHASE_SHIFT = NONE" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "CLK_FEEDBACK = 1X" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "DESKEW_ADJUST = SYSTEM_SYNCHRONOUS" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "DFS_FREQUENCY_MODE = LOW" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "DLL_FREQUENCY_MODE = LOW" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "DSS_MODE = NONE" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "DUTY_CYCLE_CORRECTION = TRUE" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "FACTORY_JF = C080" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "PHASE_SHIFT = 0" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "SIM_MODE = SAFE" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "STARTUP_WAIT = TRUE" for instance <DCM_INST> in unit <dcm1>.

Entity <dcm1> analyzed. Unit <dcm1> generated.

Analyzing module <statmach> in library <work>.


```
clear = 6'b000001
```

```
counting = 6'b001000
```

```
start = 6'b000100
```

```
stop = 6'b010000
```

```
stopped = 6'b100000
```

```
zero = 6'b000010
```

```
Module <statmach> is correct for synthesis.
```

```
Analyzing module <decode> in library <work>.
```

```
Module <decode> is correct for synthesis.
```

```
Analyzing module <cnt60> in library <work>.
```

```
Module <cnt60> is correct for synthesis.
```

```
Analyzing module <smallcntr> in library <work>.
```

```
Module <smallcntr> is correct for synthesis.
```

```
Analyzing module <hex2led> in library <work>.
```

```
Module <hex2led> is correct for synthesis.
```

```
=====
```

```
* HDL Synthesis *
```

```
=====
```

```
Performing bidirectional port resolution...
```

```
Synthesizing Unit <statmach>.
```

```
Related source file is "statmach.v".
```

```
Found finite state machine <FSM_0> for signal <current_state>.
```

```
-----
```

```
| States | 6 |
```

```
| Transitions | 15 |
```

```
| Inputs | 2 |
```

```
| Outputs | 2 |
```

```
| Clock | CLK (rising_edge) |
```

```
| Reset | RESET (positive) |
```

```
| Reset type | asynchronous |
```

```
| Reset State | 000001 |
```

```
| Encoding | automatic |
```

```
| Implementation | LUT |
```

```
-----
```

Found 1-bit register for signal <CLKEN>.

Found 1-bit register for signal <RST>.

Summary:

inferred 1 Finite State Machine(s).

inferred 2 D-type flip-flop(s).

Unit <statmach> synthesized.

Synthesizing Unit <decode>.

Related source file is "decode.v".

Found 16x10-bit ROM for signal <ONE_HOT>.

Summary:

inferred 1 ROM(s).

Unit <decode> synthesized.

Synthesizing Unit <hex2led>.

Related source file is "hex2led.v".

Found 16x7-bit ROM for signal <LED>.

Summary:

inferred 1 ROM(s).

Unit <hex2led> synthesized.

Synthesizing Unit <smallcntr>.

Related source file is "smallcntr.v".

Found 4-bit up counter for signal <QOUT>.

Summary:

inferred 1 Counter(s).

Unit <smallcntr> synthesized.

Synthesizing Unit <dcm1>.

Related source file is "C:/xst/watchver/dcm1.vhd".

Unit <dcm1> synthesized.

Synthesizing Unit <cnt60>.

Related source file is "cnt60.v".

Unit <cnt60> synthesized.

Synthesizing Unit <stopwatch>.

Related source file is "stopwatch.v".

Unit <stopwatch> synthesized.

=====

HDL Synthesis Report

Macro Statistics

ROMs : 3

16x10-bit ROM : 1

16x7-bit ROM : 2

Counters : 2

4-bit up counter : 2

Registers : 2

1-bit register : 2

=====

=====

* Advanced HDL Synthesis *

=====

Analyzing FSM <FSM_0> for best encoding.

Optimizing FSM <MACHINE/current_state/FSM> on signal <current_state[1:3]> with sequential encoding.

State | Encoding

000001 | 000

000010 | 001

000100 | 010

001000 | 011

010000 | 100

100000 | 101

Loading device for application Rf_Device from file '4v1x15.nph' in environment C:\xilinx.

Executing edif2ngd -noa "tenths.edn" "tenths.ngo"

Release 10.1 - edif2ngd K.31 (nt64)

Copyright (c) 1995-2008 Xilinx, Inc. All rights reserved.

INFO:NgdBuild - Release 10.1 edif2ngd K.31 (nt64)

INFO:NgdBuild - Copyright (c) 1995-2008 Xilinx, Inc. All rights reserved.

Writing module to "tenths.ngo"...

Reading core <tenths_c_counter_binary_v8_0_xst_1.ngc>.

Loading core <tenths_c_counter_binary_v8_0_xst_1> for timing and area information for instance <BU2>.

Loading core <tenths> for timing and area information for instance <xcounter>.

=====

Advanced HDL Synthesis Report

Macro Statistics

ROMs : 3

16x10-bit ROM : 1

16x7-bit ROM : 2

Counters : 2

4-bit up counter : 2

Registers : 5

Flip-Flops : 5

=====

* Low Level Synthesis *

=====

Optimizing unit <stopwatch> ...

Mapping all equations...

Building and optimizing final netlist ...

Found area constraint ratio of 100 (+ 5) on block stopwatch, actual ratio is 0.

Number of LUT replicated for flop-pair packing : 0

Final Macro Processing ...

=====

Final Register Report

Macro Statistics

Registers : 13

Flip-Flops : 13

=====

* Partition Report *

=====

Partition Implementation Status

No Partitions were found in this design.

=====

* Final Report *

=====

Final Results

RTL Top Level Output File Name : stopwatch.ngc

Top Level Output File Name : stopwatch

Output Format : NGC

Optimization Goal : Speed

Keep Hierarchy : NO

Design Statistics

IOs : 27

Cell Usage :

BELS : 70

GND : 2

INV : 1

LUT1 : 3

LUT2 : 1

LUT2_L : 1

LUT3 : 8

LUT3_D : 1

LUT3_L : 1

LUT4 : 37

LUT4_D : 1

LUT4_L : 4

MUXCY : 3

MUXF5 : 2

VCC : 1

XORCY : 4

FlipFlops/Latches : 17

FDC : 13

FDCE : 4

Clock Buffers : 1

BUFG : 1

IO Buffers : 27

IBUF : 2
IBUFG : 1
OBUF : 24
DCM_ADVs : 1
DCM_ADV : 1

=====

Device utilization summary:

Selected Device : 4vlx15sf363-12
Number of Slices: 32 out of 6144 0%
Number of Slice Flip Flops: 17 out of 12288 0%
Number of 4 input LUTs: 58 out of 12288 0%
Number of IOs: 27
Number of bonded IOBs: 27 out of 240 11%
Number of GCLKs: 1 out of 32 3%
Number of DCM_ADVs: 1 out of 4 25%

Partition Resource Summary:

No Partitions were found in this design.

=====

TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
GENERATED AFTER PLACE-and-ROUTE.

Clock Information:

-----+-----+-----+

Clock Signal	Clock buffer (FF name)	Load
--------------	------------------------	------

-----+-----+-----+

CLK	Inst_dcm1/DCM_INST:CLK0	17
-----	-------------------------	----

-----+-----+-----+

Asynchronous Control Signals Information:

```
-----
-----+-----+-----+
Control Signal | Buffer (FF name) | Load |
-----+-----+-----+

```

```
MACHINE/RST(MACHINE/RST:Q) | NONE(sixty/lsbcount/QOUT_1) | 8 |

```

```
RESET | IBUF | 5 |

```

```
sixty/msbclr(sixty/msbclr_f5:O) | NONE(sixty/msbcount/QOUT_0) | 4 |
-----+-----+-----+

```

Timing Summary:

Speed Grade: -12

Minimum period: 2.282ns (Maximum Frequency: 438.212MHz)

Minimum input arrival time before clock: 1.655ns

Maximum output required time after clock: 4.617ns

Maximum combinational path delay: No path found

Timing Detail:

All values displayed in nanoseconds (ns)

=====

Timing constraint: Default period analysis for Clock 'CLK'

Clock period: 2.282ns (frequency: 438.212MHz)

Total number of paths / destination ports: 134 / 21

Delay: 2.282ns (Levels of Logic = 4)

Source: xcounter/BU2/U0/q_i_1 (FF)

Destination: sixty/msbcount/QOUT_1 (FF)

Source Clock: CLK rising

Destination Clock: CLK rising

Data Path: xcounter/BU2/U0/q_i_1 to sixty/msbcount/QOUT_1

Gate Net

Cell:in->out fanout Delay Delay Logical Name (Net Name)

FDCE:C->Q 12 0.272 0.672 U0/q_i_1 (q(1))

LUT4:I0->O 11 0.147 0.492 U0/thresh0_i_cmp_eq00001 (thresh0)

end scope: 'BU2'

end scope: 'xcounter'

LUT4:D:I3->O 1 0.147 0.388 sixty/msbce (sixty/msbce)

LUT3:I2->O 1 0.147 0.000 sixty/msbcount/QOUT_1_rstpot (sixty/msbcount/QOUT_1_rstpot)

FDC:D 0.017 sixty/msbcount/QOUT_1

Total 2.282ns (0.730ns logic, 1.552ns route)

(32.0% logic, 68.0% route)

=====

Timing constraint: Default OFFSET IN BEFORE for Clock 'CLK'

Total number of paths / destination ports: 4 / 3

Offset: 1.655ns (Levels of Logic = 3)

Source: STRTSTOP (PAD)

Destination: MACHINE/current_state_FSM_FFd3 (FF)

Destination Clock: CLK rising

Data Path: STRTSTOP to MACHINE/current_state_FSM_FFd3

Gate Net

Cell:in->out fanout Delay Delay Logical Name (Net Name)

IBUF:I->O 4 0.754 0.446 STRTSTOP_IBUF (STRTSTOP_IBUF)

LUT4:I2->O 1 0.147 0.000 MACHINE/current_state_FSM_FFd3-In_F (N48)

MUXF5:I0->O 1 0.291 0.000 MACHINE/current_state_FSM_FFd3-In (MACHINE/current_state_FSM_FFd3-In)

FDC:D 0.017 MACHINE/current_state_FSM_FFd3

Total 1.655ns (1.209ns logic, 0.446ns route)

(73.0% logic, 27.0% route)

=====

Timing constraint: Default OFFSET OUT AFTER for Clock 'CLK'

Total number of paths / destination ports: 96 / 24

Offset: 4.617ns (Levels of Logic = 2)

Source: sixty/lbcount/QOUT_1 (FF)

Destination: ONESOUT<6> (PAD)

Source Clock: CLK rising

Data Path: sixty/lsbcount/QOUT_1 to ONESOUT<6>

Gate Net

Cell:in->out fanout Delay Delay Logical Name (Net Name)

FDC:C->Q 13 0.272 0.677 sixty/lsbcount/QOUT_1 (sixty/lsbcount/QOUT_1)

LUT4:I0->O 1 0.147 0.266 lsbled/Mrom_LED21 (lsbled/Mrom_LED2)

OBUF:I->O 3.255 ONESOUT_2_OBUF (ONESOUT<2>)

Total 4.617ns (3.674ns logic, 0.943ns route)

(79.6% logic, 20.4% route)

=====

Total REAL time to Xst completion: 20.00 secs

Total CPU time to Xst completion: 19.53 secs

-->

Total memory usage is 333688 kilobytes

Number of errors : 0 (0 filtered)

Number of warnings : 0 (0 filtered)

Number of infos : 1 (0 filtered)

CPLD ログ ファイルの例

次は、CPLD 合成の XST ログ ファイルの例です。

Release 10.1 - xst K.31 (nt64)

Copyright (c) 1995-2008 Xilinx, Inc. All rights reserved.

TABLE OF CONTENTS

- 1) Synthesis Options Summary
- 2) HDL Compilation
- 3) Design Hierarchy Analysis
- 4) HDL Analysis
- 5) HDL Synthesis
- 5.1) HDL Synthesis Report
- 6) Advanced HDL Synthesis
- 6.1) Advanced HDL Synthesis Report
- 7) Low Level Synthesis
- 8) Partition Report

9) Final Report

```
=====
* Synthesis Options Summary *
=====

---- Source Parameters

Input File Name : "stopwatch.prj"

Input Format : mixed

Ignore Synthesis Constraint File : NO

---- Target Parameters

Output File Name : "stopwatch"

Output Format : NGC

Target Device : CoolRunner2 CPLDs

---- Source Options

Top Module Name : stopwatch

Automatic FSM Extraction : YES

FSM Encoding Algorithm : Auto

Safe Implementation : No

Mux Extraction : YES

Resource Sharing : YES

---- Target Options

Add IO Buffers : YES

MACRO Preserve : YES

XOR Preserve : YES

Equivalent register Removal : YES

---- General Options

Optimization Goal : Speed

Optimization Effort : 1

Library Search Order : stopwatch.lso

Keep Hierarchy : YES

Netlist Hierarchy : as_optimized

RTL Output : Yes

Hierarchy Separator : /

Bus Delimiter : <>

Case Specifier : maintain
```

```
Verilog 2001 : YES

---- Other Options

Clock Enable : YES

wysiwyg : NO

=====

=====

* HDL Compilation *

=====

Compiling verilog file "smallcntr.v" in library work

Compiling verilog file "tenths.v" in library work

Module <smallcntr> compiled

Compiling verilog file "statmach.v" in library work

Module <tenths> compiled

Compiling verilog file "hex2led.v" in library work

Module <statmach> compiled

Compiling verilog file "decode.v" in library work

Module <hex2led> compiled

Compiling verilog file "cnt60.v" in library work

Module <decode> compiled

Compiling verilog file "stopwatch.v" in library work

Module <cnt60> compiled

Module <stopwatch> compiled

No errors in compilation

Analysis of file <"stopwatch.prj"> succeeded.

=====

* Design Hierarchy Analysis *

=====

Analyzing hierarchy for module <stopwatch> in library <work>.

Analyzing hierarchy for module <statmach> in library <work> with parameters.

clear = "000001"

counting = "001000"

start = "000100"

stop = "010000"

stopped = "100000"
```

```
zero = "000010"

Analyzing hierarchy for module <tenths> in library <work>.
Analyzing hierarchy for module <decode> in library <work>.
Analyzing hierarchy for module <cnt60> in library <work>.
Analyzing hierarchy for module <hex2led> in library <work>.
Analyzing hierarchy for module <smallcntr> in library <work>.
=====
* HDL Analysis *
=====

Analyzing top module <stopwatch>.
Module <stopwatch> is correct for synthesis.
Analyzing module <statmach> in library <work>.

clear = 6'b000001
counting = 6'b001000
start = 6'b000100
stop = 6'b010000
stopped = 6'b100000
zero = 6'b000010

Module <statmach> is correct for synthesis.
Analyzing module <tenths> in library <work>.
Module <tenths> is correct for synthesis.
Analyzing module <decode> in library <work>.
Module <decode> is correct for synthesis.
Analyzing module <cnt60> in library <work>.
Module <cnt60> is correct for synthesis.
Analyzing module <smallcntr> in library <work>.
Module <smallcntr> is correct for synthesis.
Analyzing module <hex2led> in library <work>.
Module <hex2led> is correct for synthesis.
=====
* HDL Synthesis *
=====

Performing bidirectional port resolution...

Synthesizing Unit <statmach>.
```

Related source file is "statmach.v".

Found finite state machine <FSM_0> for signal <current_state>.

```
-----  
| States | 6 |  
| Transitions | 15 |  
| Inputs | 2 |  
| Outputs | 2 |  
| Clock | CLK (rising_edge) |  
| Reset | RESET (positive) |  
| Reset type | asynchronous |  
| Reset State | 000001 |  
| Encoding | automatic |  
| Implementation | automatic |  
-----
```

Found 1-bit register for signal <CLKEN>.

Found 1-bit register for signal <RST>.

Summary:

inferred 1 Finite State Machine(s).

inferred 2 D-type flip-flop(s).

Unit <statmach> synthesized.

Synthesizing Unit <tenths>.

Related source file is "tenths.v".

Found 4-bit up counter for signal <Q>.

Summary:

inferred 1 Counter(s).

Unit <tenths> synthesized.

Synthesizing Unit <decode>.

Related source file is "decode.v".

Found 16x10-bit ROM for signal <ONE_HOT>.

Summary:

inferred 1 ROM(s).

Unit <decode> synthesized.

Synthesizing Unit <hex2led>.

Related source file is "hex2led.v".

Found 16x7-bit ROM for signal <LED>.

Summary:

inferred 1 ROM(s).

Unit <hex2led> synthesized.

Synthesizing Unit <smallcntr>.

Related source file is "smallcntr.v".

Found 4-bit up counter for signal <QOUT>.

Summary:

inferred 1 Counter(s).

Unit <smallcntr> synthesized.

Synthesizing Unit <cnt60>.

Related source file is "cnt60.v".

Unit <cnt60> synthesized.

Synthesizing Unit <stopwatch>.

Related source file is "stopwatch.v".

Found 1-bit register for signal <strtstopinv>.

Summary:

inferred 1 D-type flip-flop(s).

Unit <stopwatch> synthesized.

=====

HDL Synthesis Report

Macro Statistics

ROMs : 3

16x10-bit ROM : 1

16x7-bit ROM : 2

Counters : 3

4-bit up counter : 3

Registers : 3

1-bit register : 3

=====

=====

* Advanced HDL Synthesis *

=====

Analyzing FSM <FSM_0> for best encoding.

Optimizing FSM <MACHINE/current_state/FSM> on signal <current_state[1:3]> with sequential encoding.

State | Encoding

000001 | 000

000010 | 001

000100 | 010

001000 | 011

010000 | 100

100000 | 101

=====
Advanced HDL Synthesis Report

Macro Statistics

ROMs : 3

16x10-bit ROM : 1

16x7-bit ROM : 2

Counters : 3

4-bit up counter : 3

Registers : 6

Flip-Flops : 6
=====

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

Optimizing unit <stopwatch> ...

Optimizing unit <statmach> ...

Optimizing unit <decode> ...

Optimizing unit <hex2led> ...

Optimizing unit <tenths> ...

Optimizing unit <smallcntr> ...

Optimizing unit <cnt60> ...
=====

* Partition Report *

```
=====
Partition Implementation Status
-----

No Partitions were found in this design.
-----

=====

* Final Report *
=====

Final Results

RTL Top Level Output File Name : stopwatch.ngc
Top Level Output File Name : stopwatch
Output Format : NGC
Optimization Goal : Speed
Keep Hierarchy : YES
Target Technology : CoolRunner2 CPLDs
Macro Preserve : YES
XOR Preserve : YES
Clock Enable : YES
wysiwyg : NO

Design Statistics

# IOs : 28

Cell Usage :

# BELS : 413
# AND2 : 120
# AND3 : 10
# AND4 : 6
# INV : 174
# OR2 : 93
# OR3 : 1
# XOR2 : 9
# FlipFlops/Latches : 18
# FD : 1
# FDC : 5
# FDCE : 12
```



```
# IO Buffers : 28

# IBUF : 4

# OBUF : 24

=====

Total REAL time to Xst completion: 7.00 secs

Total CPU time to Xst completion: 6.83 secs

-->

Total memory usage is 196636 kilobytes

Number of errors : 0 ( 0 filtered)

Number of warnings : 0 ( 0 filtered)

Number of infos : 0 ( 0 filtered)
```


XST の命名規則

この章では、XST の命名規則について説明します。

- ・ ネットの命名規則
- ・ インスタンスの命名規則
- ・ 命名規則の制御方法

ネットの命名規則

次のXST でのネットの命名規則は、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、スラッシュ (/) またはアンダースコア (_) で区切ります。
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 の出力ネット名には **net_name_IBUF** という形式の名前を付けます。たとえば、DIN という名前の出力ネットを持つ IBUF の場合、出力 IBUF ネット名は DIN_IBUF になります。
8. OBUF の入力ネット名には **net_name_OBUF** という形式の名前を付けます。たとえば、DOUT という名前の入力ネットを持つ OBUF の場合、入力 OBUF ネット名は 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。

インスタンスの命名規則

インスタンスを命名する際は、次のインスタンスの命名規則に従ってください。前リリースの ISE® Design Suite の命名規則を使用するインスタンスを使用する場合は、XST のコマンドラインで次のオプションを使用してください。

-old_instance_names 1

次の命名規則は、優先順にリストしています。

1. インスタンス名の階層を、スラッシュまたはアンダースコアで区切って保持します。

インスタンス名が VHDL または Verilog の generate 文で生成される場合、generate 文から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
```

2. 出力信号に対しては、ステートビットを含むレジスタ インスタンスに名前を付けます。
3. 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 `_clockbuffertype` (`_BUFGP` や `_IBUFG` など) を付けます。
4. ブラック ボックスのインスタンシエーション インスタンス名を保持します。
5. ライブラリ プリミティブのインスタンシエーション インスタンス名を保持します。
6. 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。
7. 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。
8. OBUF の入力インスタンスには **instance_name_OBUF** という形式の名前を付けます。

命名規則の制御方法

次のプロパティを使用すると、命名を制御できます。これらのプロパティは、ISE の [Process Properties] ダイアログ ボックスの [Synthesis Options] ページか、コマンドラインのオプションを使用すると適用できます。詳細は、「[デザイン制約](#)」を参照してください。

- ・ 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`)

コマンド ライン モード

この章では、コマンド ラインからの XST の起動方法、コマンドおよびそのオプションについて説明します。この章は、次のセクションから構成されています。

- ・ コマンド ライン モードからの XST の実行
- ・ コマンド ライン モードのファイルの種類
- ・ コマンド ライン モードの一時ファイル
- ・ コマンド ライン モードでのスペースを含む名前
- ・ コマンド ライン モードからの XST の起動
- ・ XST スクリプトの設定
- ・ コマンド ライン モードを使用した VHDL デザインの合成
- ・ コマンド ライン モードを使用した Verilog デザインの合成
- ・ コマンド ライン モードを使用した混合言語デザインの合成

コマンド ライン モードからの XST の実行

XST は、次のいずれかの方法でコマンド ラインから実行できます。

- ・ ワークステーションで xst を実行
- ・ PC で xst.exe を実行

コマンド ライン モードのファイルの種類

XST のコマンド ライン モードでは、次のファイルが生成されます。

- ・ NGC デザイン出力ファイル (.ngc)
このファイルは、現在の出力ディレクトリに生成されます (-ofn オプションを参照)。
- ・ RTL and Technology Viewers 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

コマンドライン モードの一時ファイル

一時ファイルは、XST の一時ディレクトリに生成されます。デフォルトの一時ディレクトリは、次のとおりです。

- ・ 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 一時ディレクトリは、**set -tmpdir <directory>** で変更できます。

VHDL/Verilog コンパイル ファイルは、一時ディレクトリに生成されます。デフォルトの一時ディレクトリは、現在のディレクトリの下にある xst ディレクトリです。

XST の一時ディレクトリには、すべての XST セッションで 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るため、定期的に一時ディレクトリ内のファイルを削除することを強くお勧めします。削除しないと、一時ディレクトリ内のファイルが CPU の動作に悪影響を及ぼす場合があります。この temp ディレクトリ内のファイルは、XST では自動的に削除されません。

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。C:\my project のように、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、名前を二重引用符 (") で囲む必要があります。

この変更により、複数ディレクトリをサポートするオプション (-sd, -vlgincdir) 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、**-vlgincdir {"C:\my project" C:\temp}** のように {} でディレクトリを囲んでください。

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、新しい構文に変更することをお勧めします。

コマンドライン モードからの XST の起動

XST は、次のいずれかの方法でコマンドラインから実行できます。

- ・ XST シェル
- ・ スクリプト ファイル

XST シェルを使用した場合

xst と入力して、XST シェルを起動します。その後、コマンド名を入力してコマンドを実行します。合成を実行するには、必須のオプションとその値をすべて含めた完全なコマンドを入力する必要があります。XST では、最初に **set option_1** を入力し、次に **set option_2** を入力し、最後に **run** と入力してコマンドを実行することはできません。

オプションはすべて同時に設定する必要があるため、スクリプト ファイルの使用の方をお勧めします。

スクリプト ファイルを使用した場合

コマンドを別のスクリプト ファイルに記述して、一度に実行します。スクリプト ファイルを実行するには、UNIX または PC で次のように入力します。

```
xst -ifn in_file_name -ofn out_file_name -intstyle {silent|ise|xflow}
```

-ofn は省略可能です。省略すると、拡張子が .srp のログ ファイルが自動的に生成され、すべてのメッセージが画面に表示されます。次のオプションを使用すると、画面に表示されるメッセージの数を制限できます。

- ・ **-intstyle** silent オプション
- ・ XIL_XST_HIDEMESSAGES 環境変数
- ・ ISE® Design Suite のメッセージ フィルタ機能

詳細は、「[ログ ファイルの容量削減方法](#)」を参照してください。

次にスクリプト ファイルの使い方の例を示します。foo.scr という名前のファイルに次のテキストが記述されているとします。

```
run
-ifn ttl.prj
-top ttl
-ifmt MIXED
-opt_mode SPEED
-opt_level 1
-ofn ttl.ngc
-p <parttype>
```

このスクリプト ファイルを実行するには、次のように入力します。

```
xst -ifn foo.scr
```

また、次のように入力すると、ログ ファイルを生成できます。

```
xst -ifn foo.scr -ofn foo.log
```

スクリプト ファイルは、**xst -ifn script name** を使用するか、XST プロンプトで **script script_name** コマンドを使用して実行します。

```
script foo.scr
```

XST コマンドまたはコマンド オプションで誤った入力をする、エラー メッセージが表示され、コマンドは実行されません。たとえば、この例で示したスクリプト ファイルで、VHDL が誤って VHDL として記述されている場合、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 "-ifmt" : parameter "VHDL" is not allowed.
```

ISE Design Suite を使ってプロジェクトを作成し、ISE Design Suite から 1 度でも XST を実行したことがある場合は、XST コマンドライン モードに切り替えて、ISE Design Suite で作成されたスクリプトおよびプロジェクト ファイルを使うことができます。XST をコマンドラインから実行する場合は、プロジェクト ディレクトリで次を入力します。

```
xst -ifn <top_level_block>.xst -ofn <top_level_block>.scr
```

XST スクリプトの設定

XST スクリプトはコマンドのセットで、各コマンドのさまざまなオプションが含まれます。XST スクリプトは、次のいずれかのコマンドで設定できます。

- ・ run
- ・ set
- ・ elaborate

run コマンドを使用した場合

run コマンドは、Hardware Description Language (HDL) 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するコマンドで、スクリプト ファイルごとに 1 度だけ使用します。

run option_1 value option_2 value のように、コマンドの最初に run と記述し、その後にオプションと値を指定します。

この際、次のように改行を加えると、スクリプト ファイルが読みやすくなります。

次のようにシャープ文字 (#) を挿入すると、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。

```
run
option_1 value
# option_2 value
option_3 value
```

その場合、次の規則に従う必要があります。

- ・ 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。
- ・ オプションの前には、ダッシュ (-) を記述します。次のようなバスがあります。-ifn、-ifmt、-ofn のように記述します。
- ・ 各オプションごとに 1 つの値を指定します。値を省略できるオプションはありません。
- ・ オプションには、次のいずれかを指定します。
 - 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

run コマンド オプションおよびその値などを含む XST のオプションについては、「[XST 固有のオプション \(タイミング以外\)](#)」および「[XST 固有のオプション \(タイミング以外\) : XST コマンドラインのみ](#)」を参照してください。

UNIX のコマンドラインからは、XST のヘルプ機能を使用できます。ヘルプは、コマンドラインで「help」と入力すると表示されます。表示される情報は、サポートされているファミリ、使用可能なコマンド、オプション、およびその値です。

XST の各コマンドについて詳しい説明を表示するには、次のように入力します。

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

斜体で示された部分には、それぞれ次を入力します。

- ・ family_name : 現在のバージョンの XST でサポートされているザイリンクス デバイス ファミリを入力します。
- ・ command_name : run、set、elaborate、time のいずれかの XST コマンドを入力します。

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。次に、入力例と、実行後出力されるメッセージを示します。

```
--> help
ERROR:Xst:1356 - Help : Missing "-arch <family>". Please specify what family you want to target
available families:
  acr2
  aspartan3
  aspartan3a
  aspartan3adsp
  aspartan3e
  avirtex4
  fpgacore
  qrvirtex4
  qvirtex4
```



```

spartan3
spartan3a
spartan3adsp
spartan3e
virtex4
virtex5
xa9500x1
xbr
xc9500
xc9500x1
xpla3

```

特定のファミリで使用できるコマンドのリストを表示するには、次のように入力します。

help -arch *family_name*.

次はその例です。

help -arch virtex

Virtex®-5 の場合に run コマンドで使用可能なオプションとその値を表示するには、次のように入力します。

--> help -arch virtex5 -command run

このコマンドを実行すると、次のような情報が表示されます。

```

-mult_style           : Multiplier Style
      block / lut / auto / pipe_lut
-bufg                 : Maximum Global Buffers
      *
-bufgce               : BUFGCE Extraction
      YES / NO
-decoder_extract      : Decoder Extraction
      YES / NO
....

-ifn : *
-ifmt : Mixed / VHDL / Verilog
-ofn : *
-ofmt : NGC / NCD
-p : *
-ent : *
-top : *
-opt_mode : AREA / SPEED
-opt_level : 1 / 2
-keep_hierarchy : YES / NO
-vlginidir : *
-verilog2001 : YES / NO
-vlgcase : Full / Parallel / Full-Parallel
....

```

set コマンドを使用した場合

XST では、set コマンドが認識されます。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、「[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

set コマンドのオプション	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 (VHDL/Verilog 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)	ファイル名

elaborate コマンドを使用した場合

elaborate コマンドは、特定のライブラリ内の VHDL/Verilog ファイルを事前にコンパイルしたり、デザインを合成せずに Verilog ファイルを検証するために使用します。コンパイル処理は run コマンドで実行されるため、このコマンドの使用はオプションです。

elaborate コマンドでは、次の表に示すオプションが使用できます。これらのオプションの詳細は、「[デザイン制約](#)」を参照してください。

elaborate コマンドのオプション

elaborate コマンドのオプション	説明	値
-ifn	プロジェクト ファイル	ファイル名
-ifmt	フォーマット	vhdl、verilog、mixed
-lso	ライブラリ検索順	file_name.lso
-work_lib	コンパイルの作業ディレクトリ (最上位ブロックがコンパイルされるディレクトリ)	ディレクトリ名、work
-verilog2001	[Verilog -2001] オプション	yes、no
-vlgpath	Verilog 検索パス	ディレクトリへのパス (複数指定する場合は、スペースで区切り、二重引用符 (" ") で囲む)
-vlgincdir	Verilog インクルード ディレクトリ	ディレクトリへのパス (複数指定する場合は、スペースで区切り、{ } で囲む)

コマンドライン モードを使用した VHDL デザインの合成

次の例では、Virtex® デバイス用に記述された階層のある VHDL デザインをコマンドライン モードを使用して合成する方法を示しています。

この例では、watchvhd という VHDL デザインを使用します。このデザイン ファイルは、ISE® Design Suite をインストールしたディレクトリ内の ISEexamples\watchvhd ディレクトリにあります。

このデザインには、次の 7 つのエンティティが含まれています。

- ・ stopwatch
- ・ statmach
- ・ tenths (CORE Generator™ ソフトウェア コア)
- ・ decode
- ・ smallcntr
- ・ cnt60
- ・ hex2led

次は、コマンドライン モードを使用して VHDL デザインを合成する例です。

1. vhd_m という新しい名前のディレクトリを作成します。
2. ISE Design Suite インストール ディレクトリである ISEexamples¥watchvhd ディレクトリから次のファイルを vhd_m ディレクトリにコピーします。
 - ・ stopwatch.vhd
 - ・ statmach.vhd
 - ・ decode.vhd
 - ・ cnt60.vhd
 - ・ smallcntr.vhd
 - ・ tenths.vhd
 - ・ hex2led.vhd

7 つの VHDL ファイルで記述されたこのデザインを合成するため、プロジェクトを作成します。

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。ザイリンクスでは、混合言語プロジェクトであるかないかにかかわらず、新しいプロジェクト フォーマットを使用することをお勧めしています。この例では、新しいプロジェクト フォーマットを使用します。VHDL ファイルのみを含むプロジェクト ファイルを作成するには、別のファイルに「vhd」と記述して、その後に VHDL ファイルをリストします。ここで、ファイルの順序は重要ではありません。XST により階層が認識され、VHDL ファイルが正しい順序でコンパイルされます。

次の手順に従ってください。

1. watchvhd.prj というファイルを新規作成します。
2. 作成したファイル内に次の VHDL ファイル名を任意の順番で記述し、保存します。

```
vhdl work statmach.vhd
vhdl work decode.vhd
vhdl work stopwatch.vhd
vhdl work cnt60.vhd
vhdl work smallcntr.vhd
vhdl work tenths.vhd
vhdl work hex2led.vhd
```

3. デザインを合成するには、XST シェルまたはスクリプト ファイルを使用して次のコマンドを実行します。

```
run -ifn watchvhd.prj -ifmt mixed -top stopwatch -ofn watchvhd.ngc -ofmt NGC -p
xc5vfx30t-2-ff324 -opt_mode Speed -opt_level 1
```

-top オプションを使用して必ず最上位デザインのブロックを指定してください。

hex2led のみを個別に合成し、パフォーマンスを確認する場合は、合成する最上位エンティティを `-top` オプションで指定します。詳細は、「[XST 固有の制約 \(タイミング以外\)](#)」を参照してください。

```
run -ifn watchvhd.prj -ifmt mixed -ofn watchvhd.ngc -ofmt NGC -p xc5vfx30t-2-ff324 -opt_mode Speed -opt_level 1 -top hex2led
```

VHDL のコンパイル時に、`work` というライブラリがデフォルトで使用されます。VHDL ファイルを別のライブラリにコンパイルする場合は、ファイル名の前にライブラリ名を追加します。hex2led を `my_lib` ライブラリにコンパイルする場合、プロジェクト ファイルは次のようになります。

```
vhdl work statmach.vhd
vhdl work decode.vhd
vhdl work stopwatch.vhd
vhdl work cnt60.vhd
vhdl work smallcntr.vhd
vhdl work tenths.vhd
vhdl my_lib hex2led.vhd
```

XST でファイルの順序が正しく認識されず、次のような警告が表示される場合があります。

```
WARNING:XST:3204. The sort of the vhdl files failed, they will be compiled in the order of the project file.
```

この場合、次の手順に従います。

- ・ すべての VHDL ファイルを正しい順序で記述します。
- ・ `-hdl_compilation_order` オプションで値 `user` を `run` コマンドに追加します。

```
run -ifn watchvhd.prj -ifmt mixed -top stopwatch -ofn watchvhd.ngc -ofmt NGC -p xc5vfx30t-2-ff324 -opt_mode Speed -opt_level 1 -top hex2led -hdl_compilation_order user
```

スクリプト モードでの XST の実行 (VHDL)

多くのオプションを指定して XST のコマンドを実行する場合や、同じコマンドを繰り返し使用する場合は、スクリプト ファイルを使用すると便利です。スクリプト ファイルを使用するには次の手順に従います。

1. 現在のディレクトリにある `stopwatch.xst` という新しいファイルを開きます。以前実行した XST のシェル コマンドを作成したファイルに貼り付け保存します。

```
run -ifn watchvhd.prj -ifmt mixed -top stopwatch -ofn watchvhd.ngc
    -ofmt NGC -p xc5vfx30t-2-ff324 -opt_mode Speed -opt_level 1
```

2. `tcsh` などのシェルから次のコマンドを実行し合成します。

```
xst -ifn stopwatch.xst
```

このコマンドを実行すると、次に示すファイルが生成されます。

- ・ `watchvhd.ngc`
インプリメンテーション ツールで処理可能な NGC ファイル
- ・ `xst.srp`
XST ログ ファイル

XST のメッセージを `watchvhd.log` など異なる名前のログ ファイルに保存する場合は、次のコマンドを実行します。

```
xst -ifn stopwatch.xst -ofn watchvhd.log
```

合成の実行に多数のオプションを使用する場合などは、1 つの行に 1 つのオプションを記述するようにすると、ファイルが読みやすくなります。その場合、次の規則に従う必要があります。

- ・ 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。
- ・ 最初の行以外の行はダッシュ (-) で開始する。

値を入力するところにスペースを間違えて入れてしまうとエラーになります。ISE 8.1i サービス パック 1 以降は、ISE® Design Suite でこの余分なスペースが自動的に削除されるようになったため、ISE Design Suite で記述される XST ファイルにはスペースの影響はありません。ただし、XST ファイルを手動で修正した後に、XST をコマンドラインから実行する場合は、余分なスペースを手動で取るようにしてください。

前述のコマンド例をこの規則に従って記述すると、stopwatch.xst は次のようになります。

```
run
-ifn watchvhd.prj
-ifmt mixed
-top stopwatch
-ofn watchvhd.ngc
-ofmt NGC
-p xc5vfx30t-2-ff324
-opt_mode Speed
-opt_level 1
```

コマンドライン モードを使用した Verilog デザインの合成

次の例では、Virtex® デバイス用に記述された階層のある Verilog デザインをコマンドライン モードを使用して合成する方法を示しています。

この例では、watchver という Verilog デザインを使用します。このデザイン ファイルは、ISE® Design Suite をインストールしたディレクトリ内の ISEexamples¥watchver ディレクトリにあります。

- ・ stopwatch.v
- ・ statmach.v
- ・ decode.v
- ・ cnt60.v
- ・ smallcntr.v
- ・ tenths.v
- ・ hex2led.v

このデザインには、次の 7 つのモジュールが含まれています。

- ・ stopwatch
- ・ statmach
- ・ tenths (CORE Generator™ ソフトウェア コア)
- ・ decode
- ・ cnt60
- ・ smallcntr
- ・ hex2led

次は具体的な設定方法です。

1. vlg_m という名前のディレクトリを作成します。
2. ISE Design Suite インストール ディレクトリである ISEexamples\watchver ディレクトリから watchver ファイルを vlg_m という新たに作成されたディレクトリにコピーします。

-top オプションを使用して必ず最上位デザインのブロックを指定してください。

7 つの Verilog ファイルで記述されたこのデザインを合成するため、プロジェクトを作成します。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。このため、ザイリンクスでは、混合言語プロジェクトであるかないかにかかわらず、新しいプロジェクト フォーマットを使用することをお勧めしています。この例では、新しいプロジェクト フォーマットを使用しています。Verilog ファイルのみを含むプロジェクト ファイルを作成するには、別のファイルに「verilog」と記述して、その後に Verilog ファイルをリストします。ここで、ファイルの順序は重要ではありません。XST により階層が認識され、Verilog ファイルが正しい順序でコンパイルされます。

この例では、次の手順に従います。

1. watchver.v というファイルを新規作成します。
2. 作成したファイル内に次の Verilog ファイル名を任意の順番で記述し、保存します。

```
verilog work decode.v
verilog work statmach.v
verilog work stopwatch.v
verilog work cnt60.v
verilog work smallcntr.v
verilog work hex2led.v
```

3. デザインを合成するには、XST シェルまたはスクリプト ファイルから次のコマンドを実行します。

```
run -ifn watchver.v -ifmt mixed -top stopwatch -ofn watchver.ngc -ofmt NGC -p
xc5vfx30t-2-ff324 -opt_mode Speed -opt_level 1
```

hex2led のみを個別に合成し、パフォーマンスを確認する場合は、合成する最上位モジュールを -top オプションで指定します。詳細は、「[XST 固有の制約 \(タイミング以外\)](#)」を参照してください。

```
run -ifn watchver.v -ifmt Verilog -ofn watchver.ngc -ofmt NGC -p xc5vfx30t-2-ff324 -opt_mode
Speed -opt_level 1 -top HEX2LED
```

スクリプト モードでの XST の実行 (Verilog)

多くのオプションを指定して XST のコマンドを実行する場合や、同じコマンドを繰り返し使用する場合は、スクリプト ファイルを使用すると便利です。

スクリプト ファイルを使用するには次の手順に従います。

1. design.xst という新しいファイルを現在のディレクトリで開きます。以前実行した XST のシェル コマンドを作成したファイルに貼り付け保存します。

```
run -ifn watchver.prj -ifmt mixed -ofn watchver.ngc -ofmt NGC -p xc5vfx30t-2-ff324
-opt_mode Speed -opt_level 1
```

2. tcsh などのシェルから次のコマンドを実行し合成します。

```
xst -ifn design.xst
```

このコマンドを実行すると、次に示すファイルが生成されます。

- ・ `watchvhd.ngc`
インプリメンテーション ツールで処理可能な NGC ファイル
- ・ `design.srp`
XST スクリプト ログ ファイル

XST のメッセージを `watchvhd.log` など異なる名前のログ ファイルに保存する場合は、次のコマンドを実行します。

```
xst -ifn design.xst -ofn watchver.log
```

合成の実行に多数のオプションを使用する場合などは、1 つの行に 1 つのオプションを記述するようにすると、ファイルが読みやすくなります。その場合、次の規則に従う必要があります。

- ・ 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。
- ・ 最初の行以外の行はダッシュ (-) で開始する。

先ほどのコマンド例をこの規則に従って記述すると、`design.xst` は次のようになります。

```
run
-ifn watchver.prj
-ifmt mixed
-top stopwatch
-ofn watchver.ngc
-ofmt NGC
-p xc5vfx30t-2-ff324
-opt_mode Speed
-opt_level 1
```

コマンドライン モードを使用した混合言語デザインの合成

次の例では、Virtex® デバイス用に記述された階層のある VHDL および Verilog の混合デザインをコマンドライン モードを使用して合成する方法を示しています。

1. `vhdl_verilog` という名前のディレクトリを作成します。
2. ISE Design Suite インストール ディレクトリである `ISEexamples\watchvhd` ディレクトリから次のファイルを新たに作成した `vhdl_verilog` ディレクトリにコピーします。

- ・ `stopwatch.vhd`
- ・ `statmach.vhd`
- ・ `decode.vhd`
- ・ `cnt60.vhd`
- ・ `smallcntr.vhd`
- ・ `tenths.vhd`

3. ISE Design Suite インストール ディレクトリである `ISEexamples\watchver` ディレクトリから `hex2led.v` ファイルを `vhdl_verilog` という新たに作成されたディレクトリにコピーします。

6 つの VHDL ファイルと 1 つの Verilog ファイルで記述されたこのデザインを合成するためのプロジェクトを作成します。プロジェクト ファイルを作成するには、別のファイルに「`vhdl`」と記述してその後に VHDL ファイルをリストし、「`verilog`」と記述してその後に Verilog ファイルをリストします。ここで、ファイルの順序は重要ではありません。XST により階層が認識され、Hardware Description Language (HDL) ファイルが正しい順序でコンパイルされます。

スクリプト モードでの XST の実行 (混合言語)

多くのオプションを指定して XST のコマンドを実行する場合や、同じコマンドを繰り返し使用する場合は、スクリプト ファイルを使用すると便利です。スクリプト ファイルを使用するには次の手順に従います。

1. 現在のディレクトリにある stopwatch.xst というファイルを開きます。以前実行した XST のシェル コマンドを作成したファイルに貼り付け保存します。

```
run -ifn watchver.prj -ifmt mixed -top stopwatch -ofn watchver.ngc -ofmt NGC -p xc5vfx30t-2-ff324  
-opt_mode Speed -opt_level 1
```

2. tcsh などのシェルから次のコマンドを実行し合成します。

```
xst -ifn stopwatch.xst
```

このコマンドを実行すると、次に示すファイルが生成されます。

- ・ watchver.ngc : インプリメンテーション ツールで処理可能な NGC ファイル
- ・ xst.srp : XST スクリプト ログ ファイル

XST のメッセージを watchvhd.log など異なる名前のログ ファイルに保存する場合は、次のコマンドを実行します。

```
xst -ifn stopwatch.xst -ofn watchver.log
```

合成の実行に多数のオプションを使用する場合などは、1 つの行に 1 つのオプションを記述するようにすると、ファイルが読みやすくなります。その場合、次の規則に従う必要があります。

- ・ 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。
- ・ 最初の行以外の行はダッシュ (-) で開始する。

前述のコマンド例をこの規則に従って記述すると、stopwatch.xst は次のようになります。

```
run  
-ifn watchver.prj  
-ifmt mixed  
-ofn watchver.ngc  
-ofmt NGC  
-p xc5vfx30t-2-ff324  
-opt_mode Speed  
-opt_level 1
```