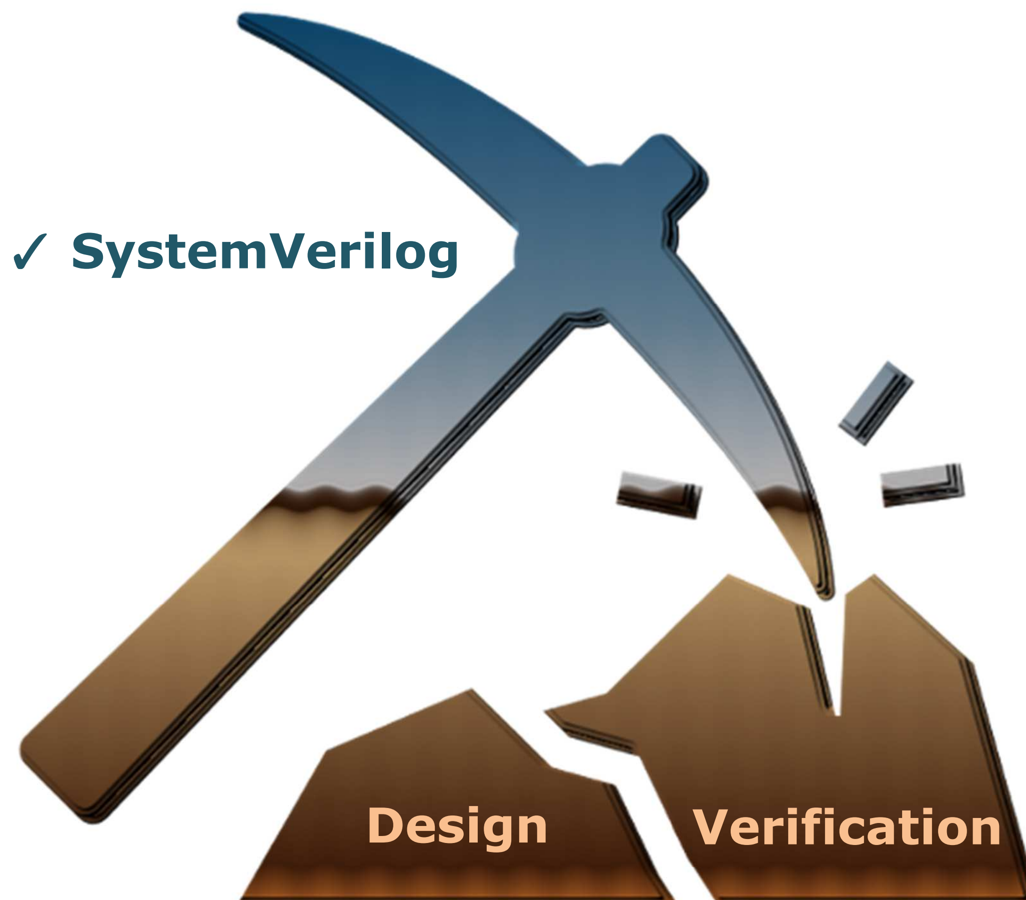
Artgraphics

SystemVerilogによる設計と検証

Document Identification Number: ARTG-TD-004-2020
Document Revision: 1.0, 2020.05.30
アートグラフィックス
篠塚一也



SystemVerilogによる設計と検証

© 2020 アートグラフィックス
〒124-0012 東京都葛飾区立石 8-14-1
www.artgraphics.co.jp

SystemVerilog for Design and Verification

© 2020 Artgraphics. All rights reserved.
8-14-1, Tateishi, Katsushika-ku, Tokyo, 124-0012 Japan
www.artgraphics.co.jp

注意事項

- 弊社の技術資料の内容の一部、あるいは全部を無断で複写、複製、転載する事は、禁じます。
- 弊社の技術資料の譲渡、転売、模倣、又は、改造等の行為を禁止致します。

はじめに

SystemVerilog の最新仕様は、2018 年 2 月 21 日に規格 IEEE Std 1800-2017（以降、LRM と略称）として公開され、実質的に Verilog HDL（以降 Verilog と略称）時代に終末を告げ、SystemVerilog の時代が到来したと言えます。SystemVerilog は Verilog の持つ曖昧性を除去すると共に Verilog が備えていない多くの機能を追加し、設計、及び検証分野での生産性向上と品質向上を齎します。特に、SystemVerilog が備えるクラスは、検証技術の再利用性を高めるためのデータタイプとして重要な役割を果たします。

SystemVerilog は、Verilog をサブセットとして位置付け、Verilog との上位互換性を維持する事を基本としています。例えば、Verilog シンタックスで記述したデザインを、変更せずに SystemVerilog の環境でコンパイルする事ができます。然し、実行結果が Verilog シミュレータの結果と完全に一致する保証はありません。理由は、Verilog の曖昧性に起因します。Verilog ではスケジューリングのセマンティックスが厳密ではないために、記述の仕方により実行結果が大きく異なります。また、同じ記述でも、使用する Verilog シミュレータにより実行結果が異なる事もあり得ます。Verilog と比較した時、SystemVerilog の大きな進歩はスケジューリングのセマンティックスを厳密に定義した事と言えます。SystemVerilog を学習する際には、先ず最初に、そのセマンティックスを理解しなければなりません。逆に言えば、スケジューリングセマンティックスを忠実に実践すれば、シミュレーション結果は期待した通りの動作になる事を意味します。本書は、全体を通してこの事実を強調します。

既に述べた様に、SystemVerilog には多くの機能が追加されました。とりわけ、SystemVerilog の豊富なデータタイプは検証作業の実践面での改革を余儀なくさせます。例えば、従来のモジュールベースのテストベンチではなく、汎用化に適した SystemVerilog クラスを使用した検証環境構築法は生産性向上と再利用可能性を促進し、検証技術をライブラリーとして蓄積する事を可能にします。従い、SystemVerilog では従来とは異なる発想が求められます。

LRM は 1300 ページに及ぶ大作であり、読破するには相当の覚悟と時間が必要です。本来、誰もが言語仕様書を読まなければならないのですが、LRM が容易に理解できる英文では書かれていない事実を考えると、少数の技術者のみが読破し得ると思えます。一方、日本国内には LRM を解説した良書が皆無である事実は、国内における SystemVerilog の普及を妨げている大きな障害の一つと言えると考えられます。

Verilog から SystemVerilog への移行、或いは設計及び検証分野の主言語として SystemVerilog を採用する事は時代の趨勢であると受け止めなければなりません。従って、ハードウェア設計検証技術者にとっては、SystemVerilog に関する実践的な知識を習得する事は、必然的な義務となっています。本書は、こうした状況を鑑みて誕生しました。

本書は、単なる SystemVerilog の解説書では無く、言語の持つ機能を基礎から解説して、実践で使用するための知識を提供する事を主眼にしています。本書は、LRM に書かれてある重要な章を殆ど含んでいるため、決して入門書とは言えないかも知れません。然し、記述スタイルは初心者を対象にしているので、本書の内容を理解する事は困難ではないと思います。

本書の構成は、LRM の構成を尊重する様に構成されているので、本書を読みながら LRM を参照する事は比較的容易です。本書の内容は、LRM のエッセンスを簡潔明瞭に解説した資料ではありますが、更に詳細な知識を得るためには LRM を参照するのが最も望ましい事です。

本書は、概要を含めて 27 章から構成され、SystemVerilog 言語全般の解説と検証機能全般の解説をカバーしています。SystemVerilog 言語全般では、Verilog との差異、SystemVerilog に追加された機能等を中心にして解説を進め、検証機能全般ではランダムスティミュラスの生成、ファンクショナルカバレッジ、アサーション、UVM を解説しています。要約すると、本書を読了後は SystemVerilog の基礎的な知識から検証技術の基礎知識までを習得する事ができます。特に、最近知られ始めている検証手法 UVM の解説は検証技術を見直す好機になると確信し

Copyright 2020 © Artgraphics. All rights reserved.

ています。UVM は、SystemVerilog が備える殆ど全ての機能を利用して構築された優れた検証パッケージです。従って、UVM を理解する事は SystemVerilog に関する理解を深める事に繋がります。UVM は決して簡単に理解できる概念ではありませんが、たとえ検証作業に関わりがない読者でも UVM に関する基礎知識を習得する事を勧めます。知識習得の努力が報われる日が必ずやって来ます。

本書は多くの章から構成されてはいますが、第 5 章までの内容を順に読んだ後は、他の章を選択して学習する事ができます。目的と必要性に応じて主題を選択して効果的に学習を進めて下さい。

本書の特徴の一つは、多くの例題でシミュレーション結果を示している事です。シミュレーション結果を示す理由は、記述した機能がどの様な効果を齎すかを正確に伝えるためです。同時に、予想外の結果を齎す事が有り得る事を示すためでもあります。即ち、単なる機能の説明に終わらず、予想される帰結を如実に示す事により、実践で遭遇し得る問題を未然に防ぐための基礎技術を研磨する機会としています。

また、紙面の都合上、一部の記述は小さな書体で記述されています。

アートグラフィックス
篠塚一也

Copyright 2020 © Artgraphics. All rights reserved.

変更履歴

日付	Revision	変更点
2020.05.30	1.0	初版。

Copyright 2020 © Artgraphics. All rights reserved.

目次

1 概要 1

1.1 SYSTEMVERILOG の歴史 1
1.2 SYSTEMVERILOG 概要 2
1.2.1 言語としての SystemVerilog 2
1.2.2 設計言語としての SystemVerilog 3
1.2.3 検証言語としての SystemVerilog 3
1.3 SYSTEMVERILOG 言語ルール 3
1.4 本書の対象者と目的 4
1.5 本書の構成 5
1.6 例題に関して 6
1.7 本書の記法 6

2 設計及び検証のためのビルディングブロック 10

2.1 設計要素 10
2.2 モジュール 10
2.3 プログラム 12
2.4 インターフェース 13
2.5 チェッカー 14
2.6 パッケージ 15
2.7 ゲートとスィッチレベルのモデリング 16
2.8 PRIMITIVE 17
2.9 CONFIGURATION 18
2.10 コンパイルユニット 18
2.11 `TIMESCALE コンパイラーダイレクティブ 19
2.12 VERILOG から SYSTEMVERILOG への移行 21
2.12.1 背景と動機 21
2.12.2 Verilog から SystemVerilog への変換 21
2.12.2.1 組み合わせ回路 21
2.12.2.2 シーケンシャル回路 23

3 データタイプ 25

3.1 データタイプとデータオブジェクト 25
3.2 LOGIC 型 26
3.3 ネット型 26
3.4 変数 28
3.5 単数系と集合系 30
3.6 ネットと変数 30
3.7 4-STATE 型 32
3.8 2-STATE 型 32
3.9 INTEGRAL データタイプ 34
3.10 REAL、SHORTREAL と REALTIME 34
3.11 VOID 型 34
3.12 CHANDLE 型 35
3.13 クラス 35
3.14 STRING データタイプ 35
3.15 イベントデータタイプ 37
3.16 TYPEDEF 39
3.17 ENUM データタイプ 40
3.18 定数 44
3.19 CONST 定数 47

Copyright 2020 © Artgraphics. All rights reserved.

3.20 CAST オペレータ 47
3.21 $CAST ダイナミック型変換 48
3.22 便利な初期値設定 49
3.22.1 リテラルの拡張 49
3.22.2 インデックス指定 50
3.23 リファレンスポインター 51

4 メンバーで構成されるデータタイプ 53

4.1 ストラクチャ 53
4.1.1 Packed ストラクチャ 55
4.1.2 ストラクチャへの値の設定 56
4.2 ユニオン 56
4.2.1 packed ユニオン 58
4.2.2 タグ付きユニオン 59
4.3 PACKED アレイと UNPACKED アレイ 59
4.3.1 packed アレイ 60
4.3.2 unpacked アレイ 60
4.3.3 アレイの操作 61
4.3.4 packed アレイのアクセス 63
4.4 ダイナミックアレイ 64
4.4.1 ダイナミックアレイのメソッド 64
4.4.2 アレイのコピー 66
4.5 ASSOCIATIVE アレイ 68
4.5.1 Associative アレイの概要 68
4.5.2 Associative アレイの要素の登録 69
4.5.3 Associative アレイの関数 70
4.5.4 Associative アレイリテラル 71
4.6 キュー 72
4.6.1 キューの概要 72
4.6.2 キューの操作 74
4.6.3 キューを操作する関数 75
4.7 アレイ情報取得関数 78
4.8 アレイ操作関数 78
4.8.1 アレイ検索関数 78
4.8.2 アレイ要素の順序を操作する関数 80
4.8.3 アレイを計算する関数 81
4.9 アレイの走査法 83

5 クラス 84

5.1 クラスの概要 84
5.2 シンタックス 85
5.3 クラスオブジェクト（クラスインスタンス） 87
5.4 クラスプロパーティ及びメソッドへのアクセス 87
5.5 コンストラクタ 88
5.6 タイプ指定のコンストラクタ呼び出し 89
5.7 STATIC クラスプロパーティ 89
5.8 STATIC クラスメソッド 91
5.9 THIS ハンドル 92
5.10 ハンドルのアレイ 92
5.11 クラスのコピー 93
5.12 クラスインヘリタンスとサブクラス 95
5.13 $CAST 98
5.14 CONST クラスプロパーティ 99

Copyright 2020 © Artgraphics. All rights reserved.

5.14.1　グローバル定数 ... 99
5.14.2　インスタンス定数 ... 99
5.15 VIRTUAL メソッド ... 99
5.16　アブストラクトクラスとピュアバーチャルメソッド ... 101
5.17　クラススコープオペレータ ... 103
5.18　メンバーへのアクセス制限 ... 104
5.19　メソッドをクラスの外に記述する方法 ... 105
5.20　パラメータによる汎用クラスの定義 ... 106
5.20.1　概要 ... 106
5.20.2　パラメータによる汎用クラスの実装 ... 107
5.20.3　パラメータによる汎用クラスの開発手順 ... 108
5.20.4 Quicksort ... 109
5.21　クラスのフォワード宣言 ... 112
5.22　インターフェースクラス ... 112
5.22.1　概要 ... 113
5.22.2　機能 ... 114
5.23　ガーベッジコレクション ... 115
5.23.1　概要 ... 115
5.23.2 automatic 変数 ... 116
5.23.3 static 変数 ... 116

6 プロセス ... 118

6.1　シミュレーションプロシージャ ... 118
6.1.1 initial プロシージャ ... 119
6.1.2 always プロシージャ ... 120
6.1.2.1　組み合わせ回路とセンシティビティリスト ... 120
6.1.2.2　シーケンシャル回路と always プロシージャ ... 120
6.1.2.3 always_comb プロシージャ ... 122
6.1.2.4 always @(*)プロシージャ ... 124
6.1.2.5 always_latch プロシージャ ... 124
6.1.2.6 always_ff プロシージャ ... 125
6.1.3 final プロシージャ ... 127
6.2　ブロック文 ... 127
6.2.1 begin-end ブロック ... 128
6.2.2 fork-join ブロック ... 129
6.2.3　ブロック名 ... 134
6.2.4 fork ブロックの効果的利用 ... 134
6.3　タイミングによる実行制御 ... 135
6.3.1　タイミングによる実行制御の概要 ... 135
6.3.2　ディレーによる制御 ... 135
6.3.3　エッジセンシティブイベント制御 ... 136
6.3.3.1　機能 ... 136
6.3.3.2　センシティビティリスト指定の注意点 ... 138
6.3.4　代入内タイミング制御 ... 140
6.3.5　レベルセンシティブイベント制御 ... 143
6.3.6　イベント制御と解除 ... 143
6.3.7　シーケンスによるイベント制御 ... 144
6.4　プロセス制御 ... 145
6.4.1 wait 文 ... 145
6.4.2 wait fork 文 ... 145
6.4.3 disable fork 文 ... 147
6.4.4 wait_order 文 ... 147
6.5　プロセスと RNG ... 148

Copyright 2020 © Artgraphics. All rights reserved.

6.6　ユーザ固有のプロセス制御 150
7　代入文 153
7.1　連続代入文 153
7.2　ビヘイビア代入文 155
7.2.1　ブロッキング代入文 155
7.2.2　ノンブロッキング代入文 156
7.2.2.1　機能 156
7.2.2.2　代入文の効果の確認 157
7.2.2.3　左辺に関する制限 160
7.3　パターン指定による代入 160
7.4　アレイパターン指定代入文 161
7.5　ストラクチャパターン指定代入文 162
7.6　UNPACKED アレイの初期化 163
8　オペレータと式 165
8.1　オペレータ 165
8.1.1　代入オペレータ 166
8.1.2　インクリメント及びデクリメントオペレータ 166
8.1.3　算術演算子 167
8.1.4　比較オペレータ 168
8.1.5　ワイルドカード比較オペレータ 169
8.1.6　logical オペレータ 171
8.1.7　bitwise オペレータ 171
8.1.8　計算オペレータ 173
8.1.9　shift オペレータ 174
8.1.10　conditional オペレータ 175
8.1.11　結合オペレータ 176
8.1.12　inside オペレータ 177
8.1.13　ビットストリームオペレータ 178
8.2　オペランド 180
8.2.1　パートセレクト 180
8.2.2　unpacked アレイ 181
8.3　タグ付きメンバーの操作 182
9　実行文 184
9.1　IF 文 184
9.1.1　全ての条件を列挙 184
9.1.2　unique-if、unique0-if 文 184
9.1.3　priority-if 文 185
9.2　CASE 文 186
9.2.1　unique-case、unique0-case 文 187
9.2.2　priority-case 文 187
9.2.3　inside オペレータと if 文及び case 文 188
9.2.3.1　if 文と inside オペレータ 188
9.2.3.2　case 文と inside オペレータ 188
9.2.1　casez と casex 189
9.3　ループ文 190
9.3.1　for 文 190
9.3.2　repeat 文 191
9.3.3　foreach 文 192
9.3.4　while 文 194
9.3.5　do-while 文 195

Copyright 2020 © Artgraphics. All rights reserved.

9.3.6 forever 文 196
9.4 RETURN 文 196
9.5 BREAK 文 197
9.6 CONTINUE 文 198

10 タスクとファンクション 200

10.1 タスク 200
10.1.1 シンタックス 200
10.1.2 ポートリスト 200
10.1.3 タスク内の記述 201
10.2 ファンクション 202
10.2.1 シンタックス 202
10.2.2 ファンクションの制限 202
10.2.3 ポートリスト 203
10.2.4 ファンクション内の記述 203
10.2.5 ファンクション内でタイミング制御を行う方法 204
10.3 引数に標準値を指定する方法 205
10.4 値を戻すファンクションの使用 206
10.5 値を戻さないファンクションの使用 207
10.6 再帰呼び出し 208
10.7 クラスのメソッドと再帰呼び出し 209
10.8 メソッド内での変数の初期化 209
10.9 引数としてのアレイ 211
10.10 インポートとエクスポート 212

11 クロッキングブロック 214

11.1 最も簡単なクロッキングブロック 214
11.2 クロッキングスキュー 215
11.3 クロッキングイベントと OBSERVED 領域 217
11.4 サイクルディレー 218

12 プロセス間の同期と交信 220

12.1 セマフォ 220
12.2 メールボックス 223
12.3 パラメータ化したメールボックス 228
12.4 名称付きイベント 228
12.4.1 概要 228
12.4.2 triggered 231
12.4.3 引数としてのイベントオブジェクト 233
12.4.4 イベント資源の解放 233
12.4.5 イベントの比較 234
12.4.6 イベントの別名 234
12.4.7 イベントによるプロセス間同期 235
12.4.7.1 検証環境の概要 235
12.4.7.2 vc_if 236
12.4.7.3 ipc_header_t 236
12.4.7.4 vc_item_t 237
12.4.7.5 vc_base_t 237
12.4.7.6 vc_generator_t 238
12.4.7.7 vc_driver_t 239
12.4.7.8 vc_collector_t 240
12.4.7.9 vc_pkg 241
12.4.7.10 dut 241
12.4.7.11 top 241
12.4.7.12 実行結果 242

Copyright 2020 © Artgraphics. All rights reserved.

13 チェッカー 243

13.1 概要 243
13.2 チェッカーインスタンス 243
13.3 自由変数 245
13.4 DUT 出力のサンプリング 246

14 プログラム 249

14.1 シンタックス 249
14.2 プログラムの特徴 250
14.3 プログラムの制御 252
14.4 シミュレーションの終了 252
14.5 テストベンチ 253

15 インターフェース 255

15.1 シンタックス 255
15.2 インターフェースの機能概要 256
15.3 ジェネリックインターフェースに依る接続 257
15.4 MODPORT 257
15.5 パラメータ化したインターフェース 260
15.6 VIRTUAL インターフェース 260
15.7 インターフェース使用例 261
15.7.1 概要 261
15.7.2 simple_if 262
15.7.3 driver 262
15.7.4 collector 263
15.7.5 dut 264
15.7.6 top 264
15.7.7 実行結果 264
15.7.8 レスポンスチェックのタイミング 265

16 パッケージ 266

16.1 シンタックス 266
16.2 パッケージの定義法 267
16.3 パッケージの使用法 267
16.4 STD パッケージ 270

17 モジュール 274

17.1 概要 274
17.2 モジュールの定義 275
17.3 ポートリスト 277
17.3.1 Verilog スタイルと SystemVerilog スタイル 277
17.3.2 ポートの方向に関するルール 278
17.4 パラメータ化したモジュール 278
17.5 トップレベルモジュール 280
17.6 モジュールインスタンス 280
17.7 モジュール記述例 281
17.7.1 一般的な記述 282
17.7.2 組み合わせ回路 283
17.7.2.1 組み合わせ回路の検証 283
17.7.2.2 組み合わせ回路の記述ルール 285
17.7.2.3 ALU 286
17.7.2.4 コンパレータ 288
17.7.2.5 デコーダー 289

Copyright 2020 © Artgraphics. All rights reserved.

17.7.2.6　エンコーダー......291
17.7.2.7　Gray コード......292
17.7.2.8　multiplexer......294
17.7.2.9　バレルシフタ......296
17.7.2.10　ファンクションユニット......297
17.7.2.11　符号付き整数の加減算......299
17.7.3　シーケンシャル回路......301
17.7.3.1　シーケンシャル回路の検証......301
17.7.3.2　シーケンシャル回路の検証手順......302
17.7.3.3　シーケンシャル回路の記述ルール......303
17.7.3.4　バイナリーカウンター......303
17.7.3.5　ラッチ......306
17.7.3.6　JK–フリップフロップ......308
17.7.3.7　データシフタ......309
17.7.3.8　ユニバーサルシフトレジスタ......311
17.7.3.9　Johnson カウンター......313
17.7.3.10　Gray カウンター......314
17.7.3.11　リングカウンター......316
17.7.3.12　Gated clock の記述例......317
17.7.3.13　インターフェースを使用するモジュール記述......319
17.7.4　FSM......321
17.7.4.1　概要......321
17.7.4.2　Moore FSM モデリング......322
17.7.4.3　Mealy FSM モデリング......325
17.7.5　ref ポートの使用......327
17.7.6　inout ポートを持つシーケンシャル回路......328
17.8　未定義モジュールの宣言......329
17.9　階層名称......331

18　ジェネレート......333

18.1　概要......333
18.2　使用法......334

19　システムタスクとシステム関数......337

19.1　$DISPLAY 及び$WRITE タスク......337
19.2　$SFORMAT と$SFORMATF......339
19.3　モニタリング......339
19.4　シミュレーション時間取得関数......340
19.5　$PRINTTIMESCALE......341
19.6　値変換......342
19.7　情報取得関数......343
19.8　ビット VECTOR システム関数......345
19.9　サンプル値を参照するための関数......346
19.10　エラー処理タスク......349
19.11　確率分布関数......350
19.12　シミュレーション制御......350
19.13　その他のシステムタスク及びシステム関数......351
19.14　コマンドラインの操作......351
19.15　VCD ファイル......353
19.15.1　VCD ファイルの指定......353
19.15.2　VCD ファイルへの記録......353
19.15.3　VCD ファイルへの記録の一時的停止と再開......354
19.15.4　VCD ファイル作成例......354

20　制約によるランダムスティミュラスの生成......356

Copyright 2020 © Artgraphics. All rights reserved.

20.1 概要 356
20.2 ランダム変数 357
20.2.1 ランダム変数の概要 357
20.2.2 rand 修飾子 358
20.2.3 randc 修飾子 358
20.2.4 rand と randc の使用例 358
20.3 乱数発生関数 359
20.4 制約 362
20.4.1 inside オペレータ 362
20.4.2 dist オペレータ 365
20.4.3 unique オペレータ 368
20.4.4 implication オペレータ 370
20.4.5 if-else 制約 371
20.4.6 foreach 制約 373
20.4.7 乱数決定順序 374
20.5 実行時に制約を定義する方法 375
20.6 ランダム変数の制御 376
20.7 制約の制御 378
20.8 RANDOMIZE()関数によるランダム変数の制御 379
20.9 条件の否定 380
20.10 ストラクチャ 381
20.11 キューに乱数を発生 382
20.12 チェッカーとしての制約 383
20.13 制約をテストケース毎に指定する方法 385
20.14 制約をクラス外部に定義する方法 386
20.15 STD::RANDOMIZE() 387
20.16 システム関数 388
20.17 RANDCASE 389

21 ファンクショナルカバレッジ 392

21.1 概要 392
21.2 カバレッジモデルの定義 393
21.2.1 シンタックス 393
21.2.2 カバーグループへの引数 395
21.2.3 サンプリングのタイミング指定 395
21.2.4 サンプリング関数 396
21.2.5 クラス外でのカバレッジ定義 397
21.2.6 カバレッジレポート 398
21.2.7 簡単な使用例 399
21.3 カバーポイントの定義 400
21.3.1 カバレッジビンの定義 401
21.3.2 固定数のビン 401
21.3.2.1 それぞれの値にビンを確保する方法 402
21.3.2.2 唯一つのビン 402
21.3.2.3 auto ビン 402
21.3.2.4 default ビン 403
21.3.2.5 illegal_bins 404
21.3.2.6 ignore_bins 404
21.3.3 カバーポイントの使用例 404
21.3.4 信号値の遷移 412
21.3.5 式のカバレッジ 414
21.3.6 制約とカバレッジ 416
21.4 クロスカバレッジ 417

Copyright 2020 © Artgraphics. All rights reserved.

21.4.1 シンタックス ... 417
21.4.2 クロスカバレッジビンの定義 ... 419
21.5 自動カバレッジ収集 ... 420
21.5.1 カバレッジ計算と検証コンポーネント ... 421
21.5.2 カバーグループの定義 ... 422
21.5.3 カバレッジのサンプリング ... 422
21.5.4 自動カバレッジ収集例 ... 422
21.5.4.1 simple_if ... 423
21.5.4.2 simple_item ... 423
21.5.4.3 simple_component ... 423
21.5.4.4 simple_driver ... 424
21.5.4.5 simple_generator ... 424
21.5.4.6 simple_collector ... 425
21.5.4.7 simple_monitor ... 425
21.5.4.8 dut ... 426
21.5.4.9 pkg ... 426
21.5.4.10 top ... 426
21.5.4.11 自動カバレッジ取集のまとめ ... 427

22 アサーション ... 428

22.1 アサーションとは何か？ ... 428
22.1.1 概要 ... 428
22.1.2 アサーションの種類 ... 428
22.2 即時実行型アサーション ... 430
22.2.1 シンタックス ... 430
22.2.2 シンプル即時実行型アサーションと遅延即時実行型アサーション ... 431
22.3 並列型アサーション ... 433
22.3.1 検証条件の評価タイミング ... 433
22.3.2 サンプリング ... 433
22.3.3 アサーションクロック ... 433
22.3.4 アサーションの式 ... 433
22.3.5 ブーリアン式（boolean expressions） ... 434
22.3.6 アサーションはマルチスレッド ... 434
22.3.7 アサーションとスケジューリング領域 ... 434
22.3.8 シンタックス ... 435
22.3.8.1 assert 文 ... 435
22.3.8.2 assume 文 ... 435
22.3.8.3 cover 文 ... 435
22.3.8.4 restrict 文 ... 436
22.3.9 implication オペレータ ... 436
22.3.9.1 機能 ... 436
22.3.9.2 vacuous pass の抑止 ... 437
22.3.10 レベルセンシティブ とエッジセンシティブ ... 438
22.3.11 アサーションスレッドダイアグラム ... 439
22.4 シーケンス ... 440
22.4.1 シーケンスの定義 ... 440
22.4.2 シーケンスの結合 ... 441
22.4.3 triggered()メソッド ... 442
22.4.4 便利なサンプル関数 ... 444
22.4.5 シーケンスの操作 ... 446
22.4.5.1 シーケンスオペレータの優先順位 ... 446
22.4.5.2 シーケンスオペレータ ... 446
22.4.6 ##m ... 447
22.4.7 ##[m:n] ... 450
22.4.8 [*m] ... 452

Copyright 2020 © Artgraphics. All rights reserved.

22.4.9 [*m:n] 454
22.4.10 [=m] 456
22.4.11 [=m:n] 457
22.4.12 [->m] 459
22.4.13 [->m:n] 461
22.4.14 seq1 and seq2 462
22.4.15 seq1 intersect seq2 464
22.4.16 seq1 or seq2 465
22.4.17 exp throughout seq 467
22.4.18 seq1 within seq2 469
22.5 プロパーティ 470
22.5.1 概要 470
22.5.2 シーケンスとプロパーティ 471
22.5.3 プロパーティオペレータ 472
22.5.4 followed-by オペレータ 473
22.5.5 always property_expr 475
22.5.6 s_always [constant_range] property_expr 479
22.5.7 nexttime property_expr 481
22.5.8 nexttime [constant_expression] property_expr 483
22.5.9 s_nexttime property_expr 485
22.5.10 s_nexttime [constant_expression] property_expr 487
22.5.11 property_expr1 until property_expr2 489
22.5.12 property_expr1 s_until property_expr2 491
22.5.13 eventually [constant_range] property_expr 493
22.5.14 s_eventually property_expr 495
22.5.15 s_eventually [cycle_delay_const_range_expression] property_expr 497
22.6 マルチクロック 499
22.6.1 マルチクロックの定義 499
22.6.1.1 ##1 499
22.6.1.2 ##0 500
22.6.2 マルチクロックの例 501

23 UVM 505

23.1 概要 505
23.1.1 UVM とは何か? 505
23.1.2 検証技術のトレンド 505
23.1.3 UVM テストベンチの構成 507
23.1.4 UVM の構成 508
23.1.4.1 uvm_pkg 508
23.1.4.2 uvm_object 508
23.1.4.3 UVM マクロ 508
23.1.5 ソースコードの準備 509
23.1.5.1 インクルード 509
23.1.5.2 uvm_pkg のインポート 509
23.2 TLM 510
23.2.1 概要 510
23.2.2 トランザクション 510
23.2.3 UVM コンポーネント間の通信 511
23.2.3.1 概要 511
23.2.3.2 put 操作 512
23.2.3.3 get 操作 516
23.2.3.4 uvm_tlm_fifo 520
23.2.3.5 analysis-ports と analysis-exports 523
23.3 代表的な UVM クラス 528
23.3.1 トランザクション関連の UVM クラス 528
23.3.2 メソドロジークラス 528

Copyright 2020 © Artgraphics. All rights reserved.

23.4 VIRTUAL インターフェース 529
23.5 UVM シミュレーション制御 529
23.5.1 シミュレーションフェーズ 529
23.5.2 シミュレーションフェーズと super.method () 532
23.5.3 サブコンポーネント作成とシミュレーションフェーズ 532
23.5.4 コンポーネント階層の情報取得関数 533
23.5.5 シミュレーションの進行 533
23.5.5.1 raise_objection()と drop_objection() 534
23.5.5.2 raise_objection()と drop_objection()の使用例 534
23.5.6 run_test 536
23.5.6.1 概要 536
23.5.6.2 使用法 537
23.5.6.3 run_test()と$finish 538
23.6 UVM クラスライブラリーの基礎 540
23.6.1 uvm_object と uvm_component 540
23.6.2 コンストラクタ 541
23.6.3 フィールドマクロ 541
23.6.3.1 トランザクション 542
23.6.3.2 メソドロジークラス 542
23.6.4 ファクトリ 543
23.6.5 コンフィギュレーションの設定変更 544
23.6.6 メッセージ機能 546
23.6.7 UVM プリンター 547
23.7 検証要素の定義 549
23.7.1 トランザクション 549
23.7.2 ドライバー 550
23.7.2.1 概要 550
23.7.2.2 ドライバー記述法 551
23.7.3 シーケンス 552
23.7.3.1 概要 552
23.7.3.2 シーケンスの定義手順 553
23.7.3.3 body()タスク 553
23.7.3.4 `uvm_do マクロと`uvm_do_with マクロ 554
23.7.3.5 raise_objection()と drop_objection() 554
23.7.3.6 pre_body()と post_body() 555
23.7.3.7 シーケンス定義例 556
23.7.4 シーケンサー 559
23.7.4.1 シーケンサーの定義 559
23.7.4.2 シーケンサーとドライバーの基本的なハンドシェーク 560
23.7.4.3 シーケンスの開始 561
23.7.5 モニターとコレクター 561
23.7.5.1 コレクターの定義 561
23.7.5.2 モニターの定義 562
23.7.6 エージェント 563
23.7.7 エンバイロンメント 564
23.7.8 テストベンチの作成 566
23.7.8.1 トップレベルの Environment の作成 566
23.7.8.2 ベーステスト 567
23.7.8.3 テスト 567
23.7.8.4 テストの選択 567
23.8 UVM による検証環境構築例 568
23.8.1 検証環境の概要 568
23.8.2 pkg_definitions.sv ファイル 570
23.8.3 alu_if 570
23.8.4 alu_item 570

Copyright 2020 © Artgraphics. All rights reserved.

23.8.5 alu_driver 570
23.8.6 alu_sequencer 572
23.8.7 alu_sequence_base 572
23.8.8 alu_test_seq1 573
23.8.9 alu_test_seq2 573
23.8.10 alu_collector 574
23.8.11 alu_monitor 575
23.8.12 alu_agent 575
23.8.13 alu_env 576
23.8.14 alu_test_base 577
23.8.15 alu_test1 577
23.8.16 alu_test2 578
23.8.17 dut 578
23.8.18 pkg 579
23.8.19 top 579
23.8.20 テストの実行 580
23.8.20.1 +UVM_TESTNAME=alu_test1 580
23.8.20.2 +UVM_TESTNAME=alu_test2 580

24 コンパイラーダイレクティブ 582

24.1 `INCLUDE 文 582
24.2 `DEFINE 文 582
24.2.1 定数を定義する場合 582
24.2.2 接頭辞及び接尾辞を持つ名称の創成 583
24.3 文字列内のパラメータ展開 584
24.4 'ENDIF 文 584
24.5 `__FILE__、`__LINE__ 585

25 シミュレーション実行モデル 587

25.1 スケジューリング領域 587
25.2 #0 ディレーの効果 588
25.2.1 スケジューリングの調節 588
25.2.2 タイミングの調節 589

26 DPI 591

26.1 概要 591
26.2 IMPORT と EXPORT 591
26.2.1 import 592
26.2.2 export 593
26.3 C LAYER 594
26.3.1 概要 594
26.3.2 4-state 型 595
26.3.3 open アレイ 597

27 補足 598

27.1 SYSTEMVERILOG 全般 598
27.2 UVM 全般 601

28 参考文献 604

Copyright 2020 © Artgraphics. All rights reserved.

3　データタイプ

SystemVerilog には多くのデータタイプが追加されましたが、それらの多くは検証分野で使用される事を目的としています。例えば、bit、byte、shortint、int、longint 等の 2-state 型は従来の 4-state 型よりもコンパクトで効率の良い検証コードを記述する事ができるという利点があります。一方、enum データタイプは設計と検証の両分野に平等に有効な機能です。例えば、parameter の代わりに、enum ラベルを使用して case 文や if 文で論理を記述する事により、RTL 論理合成の最適化機能を最大限に引き出す事ができる様になります。本章では、変数及び信号を定義する際に必要となるデータタイプを詳しく解説します。クラスもデータタイプですが、それ自身で主題を構成するので、第 5 章でクラスを解説します。

3.1　データタイプとデータオブジェクト

データタイプは、値の集合とそれらの値に適用する演算から構成されます。例えば、int は 32 ビットの符号付き整数で-2147483648 から 2147483647 までの整数値で構成され、標準的な演算が定義されています。データタイプには、SystemVerilog で予め定められたデータタイプとユーザが定義するデータタイプがあります。この章では主として前者のデータタイプを取扱います。

データタイプを使用してデータオブジェクトを宣言します。データオブジェクトは名称を付けて宣言し、名称、データタイプ、値、演算等が割り当てられる事になります。名称は、宣言したスコープ内でユニークに定まらなければなりません。また、SystemVerilog のキーワードを名称に指定する事はできません。

例 3-1　標準データタイプの例

以下の様に変数を定義する事ができます。

```
logic [31:0]  addr;
int           delay;
string        q[$];
real          map[string];
bit           p[process];
shortint      fixed[10][20];
byte          dynamic[];
```

この様に宣言すると、表 3-1 の様な効果が得られます。

表 3-1　標準データタイプの使用例の効果

宣言	説明
logic [31:0] addr	32 ビットの logic 型として宣言されています。addr は符号なしです。addr に対して標準的な演算（例えば、+, -, *, /, &, |, ^ 等）を使用する事ができます。
int delay	32 ビットの整数型です。但し、2-state です。即ち、delay は値 x、又は、z の値を取り得ません。
string q[$]	string 型のデータを持つキューです。キューに対する操作はメソッドで行います。キューはアレイの一種です。このキューの要素は、可変長である事に注意して下さい。
real map[string]	string 型のキーを持つアレイです。アレイ要素のデータは実数型です。この様なアレイを associative アレイと呼びます。
bit　p[process];	process オブジェクトをキーにする bit のアレイです。process の状態を管理するために、このアレイを使用する事ができます。
shortint fixed[10][20]	固定の大きさを持つ2次元アレイです。ここで、[10]は[0:9]の

Copyright 2020 © Artgraphics. All rights reserved.

	省略形です。
byte dynamic[]	ダイナミックなアレイです。アレイの大きさを実行時に決定します。アレイの領域も実行時に割り当てる事ができます。

■

3.2 Logic型

Logic型は4つの値（0, 1, x, z）を持ち得るデータタイプです。値0は偽の意味を持ちます。同様に、値1は真の意味を持ちます。xはunknownを意味します。 zはhigh-impedanceを意味し、接続を遮断する場合等に用います。論理合成等ではdo-not-care条件として使用します。x及び z はハードウェアには存在しませんが、ソフトウェア的に状態を表現するために使用されます。

参考 3-1

値 0 は、偽の意味を持つとして扱われますが、偽を表現するのは値 0 だけではありません。例えば、if 文に使用されている条件式が 0、x 又は z として評価されれば、条件式は偽であると判断されます。

また、値 1 だけが真として扱われる訳ではありません。例えば、if 文の条件式が 0 以外の数（整数、実数）であれば条件式は真であると判断されます。

■

SystemVerilog では、Verilog で導入された reg 型を使用せず、logic 型を使用します。reg はハードウェアのレジスタを連想させるため、極力使用しない事が勧められています。logic 型はネット型を使用する事ができる場所であれば、何処でも使用する事ができます。但し、ネットと異なり、logic 型変数は複数のドライバーを持つ事はできません。違反した場合、コンパイルエラーが発行されます。これは、logic型は複数ドライバーに対してresolutionのメカニズムを持っていないためです。ネット型の場合には、wired or や wired and という resolution のメカニズムが備わっています。

例 3-2　変数が複数のドライバーを持つ例

変数が、複数のドライバーを持つ場合、コンパイル時にエラーが発行されます。その様な例を以下に紹介します。

```
module test;
logic[1:0]    sum;
logic         a, b;

assign sum = a+b;    // error due to multiple drivers

always @(a,b)
    sum = a+b;    // error due to multiple drivers
...
endmodule
```

変数 sum が連続代入文とビヘイビア代入文の二箇所で値が割り当てられています。連続代入文とビヘイビア代入文は全く異なる構造代入文と見做されるため、sum に対して複数のドライバーが存在すると判断されます。従って、上記の記述にはコンパイルエラーが発行されます。この例の場合には、記述間違いの可能性があるので、修正が必要になります。

■

3.3 ネット型

ネットはネット型を使用して宣言し、基本的には連続代入文、又はゲートやモジュールのインスタンスに接続されて使用されます。ネット型は 4 つの値を持ち得ます。ネット型の種類

Copyright 2020 © Artgraphics. All rights reserved.

は、表3-2にまとめられています。ネットの宣言は以下のシンタックスに従います（[1]）。

```
net_declaration ::=
  net_type [ drive_strength | charge_strength ]
    [ vectored | scalared ] data_type_or_implicit [ delay3 ]
    list_of_net_decl_assignments ;
 | net_type_identifier [ delay_control ]
    list_of_net_decl_assignments ;
 | interconnect implicit_data_type [ # delay_value ]
     net_identifier { unpacked_dimension }
     [ , net_identifier { unpacked_dimension }] ;

net_type ::=
  supply0 | supply1 | tri | triand | trior | trireg | tri0 | tri1 |
  uwire | wire | wand | wor

drive_strength ::=
( strength0 , strength1 )
| ( strength1 , strength0 )
| ( strength0 , highz1 )
| ( strength1 , highz0 )
| ( highz0 , strength1 )
| ( highz1 , strength0 )
strength0 ::= supply0 | strong0 | pull0 | weak0
strength1 ::= supply1 | strong1 | pull1 | weak1
charge_strength ::= ( small ) | ( medium ) | ( large )
```

表3-2　ネット型の種類

ネット型	説明
supply0	信号値0を意味するグローバルネットです。
supply1	信号値1を意味するグローバルネットです。
tri	wireと同じ機能を持ち、tri-stateになり得るネットです。
triand	複数のドライバーを持つ場合、wired andが適用されるtriネットです。
trior	複数のドライバーを持つ場合、wired orが適用されるtriネットです。
trireg	少なくとも一つのドライバーが、0、1、xであればwireとして機能し、全てのドライバーがzであれば、以前の値を維持します（レジスタ機能）。
tri0	pulldown抵抗をモデルするネットです。ドライバーが無い時、信号値0を持ちます。
tri1	pullup抵抗をモデルするネットです。ドライバーが無い時、信号値1を持ちます。
uwire	unresolved wire又はunidriver wireを意味します。即ち、複数のドライバーを持つ事ができないネットを意味します。
wire	単なる接続機能を持ちます。
wand	複数のドライバーを持つ場合、wired andが適用されるネットです。
wor	複数のドライバーを持つ場合、wired orが適用されるネットです。

ネットの宣言においてデータタイプが省略されるとlogic型が仮定されます。ネット型が明示的に指定されないと、標準ネット型が仮定される場合があります。標準ネット型の設定は、コンパイラーダイレクティブ`default_nettypeを使用して、以下の様に行います。

```
`default_nettype default_nettype_value

default_nettype_value ::= wire | tri | tri0 | tri1 | wand |
 triand | wor | trior | trireg | uwire | none
```


Copyright 2020 © Artgraphics. All rights reserved.

また、typedef の機能を利用すると、まだ存在しないデータタイプを暫定的に定義する事ができます。相互に参照しあう 2 つのデータタイプを定義する際には、この機能が必要になります。この機能を forward typedef 宣言と呼びます。C++の forward 宣言に相当します。

例 3-17 forward typedef 宣言例

以下の記述では、クラス component_t において、未定義のクラス object_t を参照しています。このままでは、コンパイルエラーとなるため、未定義のクラスを暫定的に定義する目的で typedef 文が使用されています。然し、クラス object_t はコンパイルユニットの何処かで定義されなければなりません。この例では、component_t の後に定義されています。

```
typedef class object_t;

class component_t;
object_t  object;
// ...
endclass
...
class object_t;
// ...
endclass
```


■

3.17 Enum データタイプ

Enum データタイプは関連する値を持つコンスタントを定義する命令です。値を明示的に設定する事ができますが、自動的に設定させる事もできます。enum は以下の様なシンタックスを持ちます（[1]）。

```
enum [ enum_base_type ]
    { enum_name_declaration { , enum_name_declaration } }
    { packed_dimension }

enum_base_type ::=
    integer_atom_type [ signing ]
  | integer_vector_type [ signing ] [ packed_dimension ]
  | type_identifier [ packed_dimension ]

enum_name_declaration ::=
    enum_identifier [ [ integral_number [ : integral_number ] ] ]
      [ = constant_expression ]
```

enum_base_type には integral データタイプ等の整数型を指定します。enum_name_declaration には enum に属する定数を指定します。本書では、これらの定数を enum ラベルと呼ぶ事にします。尚、データタイプが省略されると、int 型が仮定されます。

C++の enum 機能と類似していますが、異なる点が多くあります。例えば、SystemVerilog の enum データタイプには integral type（logic、bit、byte、shortint, int 等々）を指定する事ができます。また、連続したシーケンスを持つラベル名称を簡単に生成する事が出来ます。更に、enum データタイプには、クラスの様に標準的なメソッドが定義されています。先ず、enum の使用例を示します。

例 3-18 enum 型の使用例

enum データタイプを持つ変数を以下に様に宣言する事ができます。これらの宣言は、enum から直接定義しているため、anonymous enum と呼ばれます。

Copyright 2020 © Artgraphics. All rights reserved.

```
enum {GREEN, YELLOW, RED} light1, light2;
enum bit[1:0] { READY, READ, WRITE } state, next;
enum { bronze=3, silver, gold } medal;
enum bit[3:0] { a=4'h3, b=4'h7, c }  alphabet;
```

これらの宣言により表3-10の様な定義が導かれます。

表3-10　enumデータタイプとして宣言された変数の型

変数	enum_base_type	設定される内容
light1 light2	int	int 型の enum データタイプです。GREEN=0、YELLOW=1、RED=2となります。
state next	bit[1:0]	bit型のenumデータタイプです。サイズは2ビットです。READY=2'b00、READ=2'b01、WRITE=2'b10となります。
medal	int	int 型の enum データタイプです。bronze=3、silver=4、gold=5となります。
alphabet	bit[3:0]	4 ビットの enum データタイプです。a=4'h3、b=4'h7、c=4'h8となります。

ここで、注意が必要です。変数はenum_base_typeの標準的な初期値により初期化が行われます。例えば、light1はint型なので、light1の初期値は0となります。従って、たまたまlight1の初期値はGREENと一致します。然し、いつもこの様に運が良いとは限りません。

medalはint型なので、初期値は0となります。然し、0に該当するenumラベルが無いため、実質的に無効な値が設定されている事になります。この問題を避けるためには、以下の何れかの対策が必要です。

- 変数を明示的に初期化する。例えば、medal=bronzeを追加する。
- 最初のenumラベルの値を0に設定する。

以下は安全策を考慮した記述例です。

```
enum {GREEN, YELLOW, RED} light1=GREEN, light2=RED;
enum bit[1:0] { READY, READ, WRITE } state=READY, next=READ;
enum { bronze=3, silver, gold } medal=bronze;
enum bit[3:0] { a=4'h3, b=4'h7, c }  alphabet=c;
```

特に、FSM の状態を enum で表現する場合、現在を表す状態変数が自動的に初期化される様にデザインしなければ、FSMは正しく動作しないので注意して下さい。
■

通常、anonymous enum の様に使用せずに、名称を明示的に指定した enum データタイプを定義します。そうする事により、グローバルなデータタイプとして宣言する事ができます。

例3-19　enum型のデータタイプの宣言例

次の様にtypedef文を使用して、enum型のデータタイプを宣言する事ができます。

```
typedef enum {GREEN, YELLOW, RED} controller_e;

controller_e  light1=GREEN,
              light2=RED;
```

この様に宣言すると、データタイプとしてcontroller_eを何処でも使える様になります。

Copyright 2020 © Artgraphics. All rights reserved.

表 3-12　可変長リテラルの種類

可変長リテラル	機能
'0	全てのビット位置に 0 を設定します。
'1	全てのビット位置に 1 を設定します。
'x、又は、'X	全てのビット位置に x を設定します。
'z、又は、'Z	全てのビット位置に z を設定します。

例えば、以下の様にすると、v の全てのビット位置に 1 が設定されます。

```
v = '1;
```

これは、次の様な記述の簡略形式と考えられます。下記の右辺は、パターン指定の代入文として知られています。

```
v = '{ default:1 };
```

この形式によると、次の様にビット位置の指定が出来るので、柔軟性があります。

```
v = '{ 3:0, default:1 };
```

可変長リテラルは、あくまで、全てのビット位置に同じ値を設定する場合に適しています。

例 3-28　可変長リテラルの使用例

可変長リテラルは値を設定する対象のビット幅が、大きいか、又は変化する可能性がある場合に非常に有効になります。以下は、4 入力の multiplexer の記述ですが、入力のビット幅が 4 以外に変更する場合にも容易に対応する事ができます。

```
module mux(input [3:0] a,b,c,d,[1:0] sel,output [3:0] out);

assign out = (sel == 0) ? a : (sel == 1) ? b :
             (sel == 2) ? c : (sel == 3) ? d : 'x;

endmodule
```

■

3.22.2　インデックス指定

整数型の変数に初期値をビット単位で設定したい場合があります。この様な場合、SystemVerilog ではビット位置を指定する機能があります。以下に、その例を紹介します。

例 3-29　インデックス指定による初期化

int 型の変数にビット指定する場合、以下に示す様に詳細な設定が可能です。ビット位置ごとに値を設定する事ができますが、全体としての標準値をキーワード default を用いて指定する事ができます。

```
module test;
int    state = '{ 31:0, 3:0, default:1 };
logic [15:0] value = '{ 15:1, 14:0, 0:1, default: 'bz };

   initial begin
      $display("state=%b",state);
```


Copyright 2020 © Artgraphics. All rights reserved.

```
        $display("value=%b",value);
    end
endmodule
```

実行結果は以下の様になります。

```
state=0111111111111111111111111110111
value=10zzzzzzzzzzzzz1
```

■

3.23　リファレンスポインター

SystemVerilog では、ポートの属性として input、inout、output に加えて ref タイプを追加しました。input、inout、output のポートの場合、モジュール、タスク、ファンクションの呼び出しにおいて、値のコピー操作が発生するため、値による引き渡しと呼ばれます。

一方、ref タイプのポートでは、ポート自身が直接引き渡されるため、値のコピー動作は発生しません。大きなアレイを引数として指定している場合には、ref タイプのポートは実行効率において優れています。大きなアレイを指定していなくても ref タイプのポートが必要な状況は発生します。以下に、典型的な例を紹介します。

例 3-30　ref タイプのポートが必要な例

先ず、以下の様なモジュールを仮定します。このモジュールでは、ポート a と b をクロックに同期してスワップします。ポート a と b は入力であり、且つ出力であるため、a と b は inout ポートとして宣言されています。然し、SystemVerilog のルールから、inout ポートはネットになります。

```
module swap(input logic clk,inout logic a,b);

   always @(posedge clk) begin
      a <= b;
      b <= a;
   end

endmodule
```

a と b は wire であるため、always プロシージャでドライブする事はできない。

ところが、ネットは always プロシージャで値を設定する事が出来ないため、この記述では構文エラーが発生します。

一方、inout ポートは var になり得ないので、a と b を var として指定する事は出来ません。残された選択肢は ref ポートです。ref ポートで以下の様に、問題を解決できます。

```
module swap(input logic clk,ref logic a,b);

   always @(posedge clk) begin
      a <= b;
      b <= a;
   end

endmodule
```

ref ポートを使用せずに、問題を回避する為には、以下の様に中間的に変数を使用しなければなりません。尚、この記述は RTL です。

Copyright 2020 © Artgraphics. All rights reserved.

5.14 const クラスプロパーティ

static、local 等の属性と同様に、クラスに定義されているプロパーティには、const 属性を付加する事ができます。プロパーティが const 属性を付加されると、read-only になります。然し、クラスのインスタンスは、元来、ダイナミックに作られる性質を持つ事から、クラスには 2 種類の const プロパーティがあります。それらは、グローバル定数とインスタンス定数です。

5.14.1 グローバル定数

グローバル定数は、宣言時に初期化処理を伴う定数で、他の場所ではグローバル定数に値を設定する事ができません。以下の例を文献[1]から引用しました。プロパーティ max_size がグローバル定数として宣言されています。宣言時に値を設定しているので、Jumbo_Packet の全てのインスタンスにおいて max_size の同じ値を共有する事ができます。

```
class Jumbo_Packet;
    const int max_size = 9 * 1024; // global constant
    byte payload [];

    function new( int size );
        payload = new[ size > max_size ? max_size : size ];
    endfunction

endclass
```

5.14.2 インスタンス定数

一方、インスタンス定数はクラスのインスタンス毎に初期化設定をする定数です。初期化処理は実行時に行われますが、new コンストラクタ内でのみ初期化が可能です。逆に言えば、インスタンス定数の初期化を new コンストラクタ内で行わなければなりません。

例えば、以下の様な例が、文献[1]に示されています。プロパーティ size は const 宣言されていますが、初期化されていません。この様な場合には、new コンストラクタ内で size の初期化をしなければなりません。

```
class Big_Packet;
const int size; // instance constant
byte payload [];

    function new();
        size = $urandom % 4096; //one assignment in new -> ok
        payload = new[ size ];
    endfunction

endclass
```

参考 5-1

インスタンス定数では、インスタンス毎に定数の値が異なる事になります。そして、グローバル定数の場合には、通常、static 属性を伴います。

5.15 Virtual メソッド

SystemVerilog の virtual メソッドは、C++でいう virtual ファンクションと同じ概念です。ベースクラスに virtual メソッド vm を定義すると、継承されたクラスのオブジェクトに一致する vm を呼び出します。言い換えると、クラスハンドルがベースクラスのタイプでも、ハンドルが継承されたクラスのインスタンスを示していれば、継承されたクラスの vm を呼び出します。図 5-4 を参考にして下さい。ハンドル p は subclass のオブジェクトを示しているので、p.vm()は subclass::vm()を呼び出します。

Copyright 2020 © Artgraphics. All rights reserved.

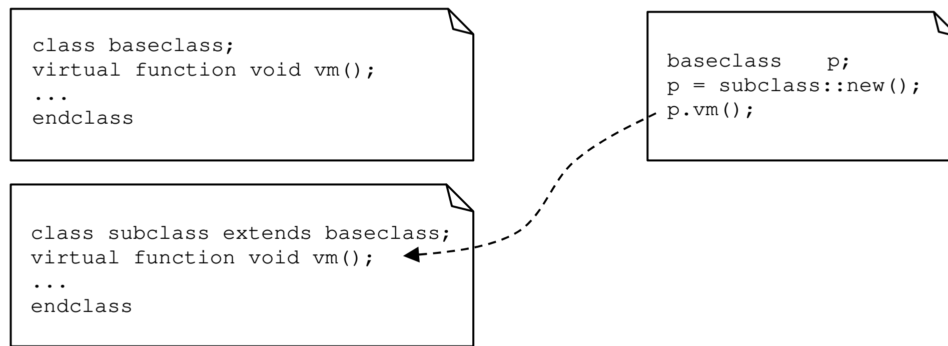


図 5-4　virtual メソッドの効果

例 5-10　virtual メソッドの例

ベースクラスを次の様に定義します。ベースクラスには、2 つのメソッドが定義されています。print()メソッドは virtual ですが、print_class()メソッドは virtual ではありません。

```
class base_transaction_t;                          // base class

virtual function void print;                       // virtual
   $display("base_transaction_t no members.");
endfunction

function void print_class;                         // non-virtual
   $display("base_transaction_t");
endfunction
endclass
```

サブクラスを次の様に定義します。サブクラスでは、ベースクラスの 2 つのメソッドを書き換えています。

```
class transaction_t extends base_transaction_t;    // subclass
rand bit [15:0]  addr, data;

function new(bit [15:0] a=0,d=0);
   addr = a;
   data = d;
endfunction

function void print;
   $display("transaction_t has two members: addr=%0d data=%0d",
      addr,data);
endfunction

function void print_class;
   $display("transaction_t");
endfunction

endclass
```

virtual メソッドの動作を確認するために、テストベンチを以下の様に定義します。

Copyright 2020 © Artgraphics. All rights reserved.

```
event  ev1, ev2;
ev1 = ev2;
```

により、ev1 と ev2 が同じイベントとなります。

例 12-11　別名の使用例

イベントの別名定義の例を以下に示します。

```
module test;
event  ev1, ev2;

   initial begin
      @ev1 $display("@%0t: ev1 released",$time);
   end

   initial begin
      #10;
      ev2 = ev1;
      ->ev2;          // trigger ev1 through ev2
   end
endmodule
```

上記の記述では、@ev1 でイベントの解除を待ちますが、->ev2 で解除を試みています。実行結果は以下の様になります。

```
@10: ev1 released
```

■

12.4.7　イベントによるプロセス間同期

ここでは、トランザクションを作るプロセスと DUT をドライブするドライバープロセスをイベントにより同期を取って制御をする例を紹介します。検証環境構築には、クラスを使用します。

12.4.7.1　検証環境の概要

図 12-5 の様な検証環境を構築して DUT を検証します。

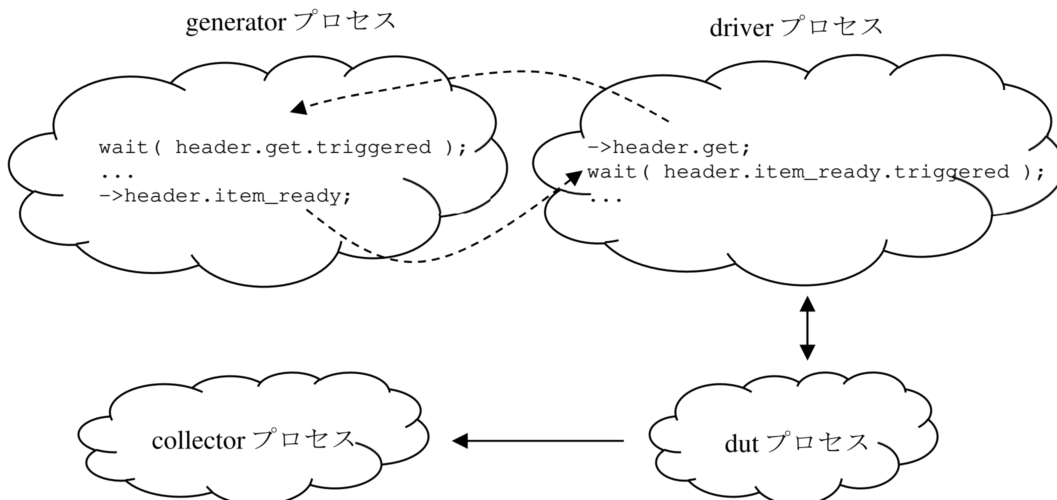


図 12-5　イベントによるプロセス間同期を使用した検証環境例

Copyright 2020 © Artgraphics. All rights reserved.

プロセス間の同期には、安全なtriggeredメソッドを使用します。DUTとしては、簡単な加算器を使用します。検証環境で使用する検証要素は、表12-4の通りです。

表12-4　検証環境の要素

クラス及びDUT	説明
vc_if	DUTをドライブするために使用するインターフェース。
ipc_header_t	プロセス間通信に必要な情報を管理するクラス。
vc_item_t	トランザクションを示すクラス。
vc_base_t	検証コンポーネントのベースクラス。
vc_generator_t	トランザクションを生成するジェネレータを定義するクラス。
vc_driver_t	DUTをドライブするドライバーを定義するクラス。
vc_collector_t	DUTのレスポンスをサンプリングするコレクターを定義するクラス。
vc_pkg	検証要素をグルーピングするためのパッケージ。
dut	2ビット加算器（組み合わせ回路）で、仕様は以下の通り。 `module dut(input logic [1:0] a,b,output co,[1:0] sum)`
top	検証環境のトップモジュール。

以下、順に検証環境要素を解説しますが、最も重要な解説はgeneratorプロセスとdriverプロセスのeventによる同期手順です。単純に、triggeredメソッドを使用するだけでは、正しい同期を達成する事はできません。トランザクションに処理状態を記録する必要があります。

12.4.7.2　vc_if

DUTをドライブするためにvirtualインターフェースが必要なので、インターフェースを以下の様に定義します。この定義では、DUTが必要な信号とクロック信号だけを含んでいます。

```
interface vc_if(input bit clk);
logic [1:0]   a, b, sum;
logic         co;
endinterface
```

12.4.7.3　ipc_header_t

このクラスは、プロセス間交信に必要な資源を定義する役割を持ちます。定義内容は、表12-5の示す通りです。

```
class ipc_header_t;
vc_item_t item;
event     get,
          item_ready;

function new;
   item = new;
endfunction

endclass
```

表12-5　ipc_header_tクラスのプロパティ

プロパティ	機能
item	トランザクションを示すオブジェクトのハンドルです。 generatorプロセスは、itemにトランザクションを準備します。
get	driverプロセスは、トランザクションが必要になると、このイベントを解除します。このイベントが解除されると、generatorプロセスは実

Copyright 2020 © Artgraphics. All rights reserved.

```
    red = on;
    ...
```

従って、変数 red に対する割り当てに関して、$time==0 に於いて、競合状態が発生する可能性があるため、ここに示す例では initial プロシージャによる初期化を省きました。

実際問題として、完全な並列処理シミュレータの環境では、initial プロシージャの初期化が存在すると実行する度に異なる結果を得ます。寧ろ、initial プロシージャを避けて、以下の様に変数の宣言時に初期化をする方が安全だと思えます。

```
logic  clock,
       red = off,
       amber = off,
       green = off;
```

この LRM の例は、Verilog 時代から知られています。この例は Verilog と SystemVerilog の差異を理解する適切な例とも考えられます。
■

17.7.2　組み合わせ回路

組み合わせ回路として以下の様な回路の記述例を紹介します。

- ALU
- コンパレータ
- デコーダー
- エンコーダー
- Gray コード変換（Gray コードをバイナリーコードに変換）
- multiplexer
- バレルシフタ
- ファンクションユニット
- 符号付整数の加減算

17.7.2.1　組み合わせ回路の検証

組み合わせ回路の例を示す前に、組み合わせ回路の検証法について説明をしておきます。

17.7.2.1.1　組み合わせ回路のセンシティビティリスト

組み合わせ回路の出力は入力で決定されます。従って、組み合わせ回路を検証する場合、入力値の変化に対してだけ結果を確認すれば良い事になります。組み合わせ回路の入力を（i1,…,in）とすると、テストベンチのセンシティビティリストは、@(i1,…,in)となります。

例えば、以下の様な簡単な組み合わせ回路を仮定します。この回路をテストする場合、結果の確認は入力 a と b の組が変化するケースだけを考慮すれば十分です。信号 out の変化に配慮する必要はありません。即ち、センシティビティリストは@(a,b)であり、@(a,b,out)ではありません。

```
module dut(input a,b,output logic out);
assign out = a | b;
endmodule
```

一般的には、テストベンチを以下の様に記述します。

```
module test;
logic  a, b, out;
```


Copyright 2020 © Artgraphics. All rights reserved.

```
dut DUT(.*);

    initial begin
        for( int i = 0; i < 4; i++ )
            #10 {a,b} = i;
    end

    initial
        $monitor("@%0t: a=%b b=%b out=%b",$time,a,b,out);
endmodule
```

@(a,b,out)

このテストベンチは正しい結果をプリントしますが、先程の前提条件に違反しています。即ち、テストベンチで使用している$monitor タスクでは、センシティビティリストが、@(a,b,out)となっているので、DUT の出力 out を余分に考慮しています。out は、a と b に依存しているので、センシティビティリストに指定する必要がありません。寧ろ、 out を指定しているので正しい動作の検証になっていないと言えます。

$monitor タスクを使用すると、組み合わせ回路を正しく検証できません。例えば、組み合わせ回路が、以下の様に記述されていると仮定します。

```
module bad_dut(input a,b,output logic out);

always @(a,b)
    out <= a | b;

endmodule
```

組み合わせ回路として正しい記述法ではない。

この様に記述されていても、先程の$monitor タスクは正しい結果をプリントしてしまいます。然し、この様な記述スタイルは、RTL の組み合わせ回路としては正しくありません。即ち、$monitor タスクは、Postponed 領域で動作するため、組み合わせ回路の出力を検証するためのタイミングを考慮する能力に欠けています。

17.7.2.1.2　組み合わせ回路を検証するタイミング

組み合わせ回路を検証する場合、以下の点に注意する必要があります。

- $monitor タスクによる組み合わせ回路の検証は、Postponed 領域で実行するため検証するタイミングが遅すぎる。
- 組み合わせ回路は、本来、Active 領域で動作する。従って、組み合わせ回路の検証を Inactive 領域で行うのが最適である。
- Inactive 領域で検証を行えば、bad_dut の記述が正しくない事も判断する事ができる。

組み合わせ回路を検証するために使用する事ができる手段を表 17-4 にまとめます。

表 17-4　組み合わせ回路を検証するの手段

検証手段	領域	bad_dut 判定能力	説明
$display	Active	有	競合状態があるため、DUT のレスポンスを正しくサンプリングする事が出来ない。
#0 $display	Inactive	有	DUT は Active 領域で動作するので、$display タスクにより DUT のレスポンスを安定した状態で、サンプリングする事が出来る。従って、テストベンチの動作は正しい。
program	Reactive	無	DUT からのレスポンスを正しくサンプリングする事ができる。但し、組み合わせ回路を検証
$monitor	Postponed	無	

Copyright 2020 © Artgraphics. All rights reserved.

```
    end

    initial forever @(a,b,subtract) begin
       #0 $display("@%3t: %2d %s (%2d) = %3d",
              $time,a,subtract?"-":"+",b,
              overflow?signed'({co,sum}):sum);
    end
endmodule
```

実行結果は、以下の様になります。負の数に対して加算、及び減算が正しく行われている事が分かります。

```
@ 10: -6 + ( 4) =  -2
@ 20: -4 + ( 1) =  -3
@ 30:  1 + ( 3) =   4
@ 40:  4 + ( 2) =   6
@ 50: -8 + ( 7) =  -1
@ 60:  7 + ( 7) =  14
@ 70:  7 + ( 0) =   7
@ 80:  2 - ( 7) =  -5
@ 90:  4 - ( 1) =   3
@100:  6 + (-2) =   4
@110: -8 - ( 0) =  -8
@120:  4 - (-8) =  12
```

■

17.7.3　シーケンシャル回路

シーケンシャル回路として以下の様な回路の記述例を紹介します。

- バイナリーカウンター
- ラッチ
- JK－フリップフロップ
- データシフタ
- ユニバーサルシフトレジスタ
- Johnson カウンター
- Gray カウンター
- リングカウンター
- Gated clock

記述例を示す前に、シーケンシャル回路を記述する際のルールと検証法を説明しておきます。

17.7.3.1　シーケンシャル回路の検証

シーケンシャル回路の記述例を紹介する前に、シーケンシャル回路の検証をするタイミングについて説明をしておきます。組み合わせ回路と異なり、シーケンシャル回路ではノンブロッキング代入文が使用されるため、シーケンシャル回路が生成する出力をサンプリングするタイミングは複雑になります。即ち、出力信号値が安定した後に、出力信号値のサンプリングが行わなければなりません。

17.7.3.1.1　シーケンシャル回路を検証するタイミング

シーケンシャル回路は、クロック等の信号で同期をとり動作します。従って、シーケンシャル回路への入力は、同期信号のイベントが発生するまでに安定した状態にあれば良いので、入力に関するタイミングを容易に制御できます。一方、シーケンシャル回路は、表 17-9 の様な種類の動作記述を含むため、シーケンシャル回路の出力をサンプリングするタイミングは複雑になります。

Copyright 2020 © Artgraphics. All rights reserved.

表 17-9　シーケンシャル回路の動作の代表的スケジューリング領域

シーケンシャル回路における動作	スケジューリング領域
assign q = counter;	Active 領域

always @(count or start)
    if(start)
        new_count = 1;
    else
        new_count = count+1;
	Active 領域

q <= d;	NBA 領域

このため、シーケンシャル回路の出力をサンプリングするタイミングは、NBA 領域よりも後でなければ競合状態を避ける事は出来ません。

17.7.3.1.2　シーケンシャル回路のレスポンス

NBA 領域よりも後に動作する命令を書く事ができる SystemVerilog の機能としては、以下の様な選択肢があります。

- Observed 領域
- Reactive 領域
- Postponed 領域

これらの領域で出力をサンプリングするためには、対応する領域で動作する様に検証コードを記述しなければなりません。以下の表 17-10 は、それぞれの領域を具体的に示しています。

表 17-10　競合状態を避ける事ができる代表的スケジューリング領域

SystemVerilog 機能	記述例	スケジューリング領域
クロッキングブロック	clocking cb @(posedge clk); endclocking always @cb …	Observed 領域
program checker	program test; ... endprogram	Reactive 領域
$monitor $strobe	$monitor("@%0t: a=%b b=%b", $time,a,b);	Postponed 領域

program ブロックは Reactive 領域で実行するので、シーケンシャル回路の出力のサンプリングには適しています。然し、program ブロックには特異性があるため、全ての状況において使用可能なわけではありません。例えば、UVM を適用する検証環境では、program ブロックを使用する事ができません。

同様に、チェッカーのインスタンスは Reactive 領域で実行するので、シーケンシャル回路の出力のサンプリングには適しています。

$monitor タスクは使用法が簡単である反面、動作するのは Postponed 領域であるので、検証するタイミングが遅すぎるという欠点があります。望ましいサンプリング法は、クロッキングブロックを用いる方法であると言えます。この手法は、UVM と両立するので、幅広い検証に適用する事ができます。

17.7.3.2　シーケンシャル回路の検証手順

クロッキングブロックを使用してシーケンシャル回路を検証する際、以下の手順に従う事を勧めます。

Copyright 2020 © Artgraphics. All rights reserved.

```
program test(simple_if.TEST mp);

initial
   fork
      begin mp.load = 1; mp.d = 9; #15 mp.load = 0; end
      begin #20 mp.up = 1; #40 mp.up = 0; end
      begin #100 mp.pc = 1; #20 mp.pc = 0; mp.up = 1; end
   join

initial forever @(posedge mp.clk)
   $display("@%3t: pc=%b load=%b up=%b d=%2d q=%2d(%b) qn=%2d(%b)",
      $time,mp.pc,mp.load,mp.up,mp.d,
      mp.q,mp.q,mp.qn,mp.qn);

endprogram
```

実行結果は、以下の様になります。

```
@ 10: pc=0 load=1 up=0 d= 9 q= 9(1001) qn= 6(0110)
@ 30: pc=0 load=0 up=1 d= 9 q=10(1010) qn= 5(0101)
@ 50: pc=0 load=0 up=1 d= 9 q=11(1011) qn= 4(0100)
@ 70: pc=0 load=0 up=0 d= 9 q=10(1010) qn= 5(0101)
@ 90: pc=0 load=0 up=0 d= 9 q= 9(1001) qn= 6(0110)
@110: pc=1 load=0 up=0 d= 9 q= 0(0000) qn=15(1111)
@130: pc=0 load=0 up=1 d= 9 q= 1(0001) qn=14(1110)
```

■

17.7.4 FSM

FSMは、有限個の異なる状態を持つシーケンシャル回路です。カウンターは、FSMの特殊な場合で、状態と出力が同一で、状態に対する選択肢はありません。カウンターの状態は、一定のルールに従い変化して行きます。

17.7.4.1 概要

FSMでは、一般に、入力と現在の状態から次の状態と出力が決定されます。FSMとしては、表17-16に示す様な二種類のタイプが知られています。

表17-16 Moore FSMとMealy FSM

FSMのタイプ	回路の動作
Moore	MooreタイプのFSMでは、出力は現在の状態にのみ依存します。 ① 組み合わ回路により、入力と現在の状態から次の状態を計算してレジスタに保存します。 ② 出力は、現在の状態から組み合わせ回路で計算します。 ③ 出力は状態に対応します。
Mealy	MealyタイプのFSMでは、出力は入力と現在の状態に依存します。 ① 出力は入力の変化に対応して即座に変化します。従って、出力は、クロックに対して非同期となります。 ② 出力は、状態の変遷に対応します。

MealyタイプのFSMでは、出力を状態の変遷に対応させる事ができるため、MooreタイプのFSMよりも少ない状態数で済む場合が多いと云われています。例えば、ビットシーケンスを入力するFSMにおいて、ビット1を2回連続して入力すると1を出力するとします。それぞれのFSMの状態遷移図は、図17-15の様になります。Moore FSMの方が状態数を余計に必要とする事が分かります。

Copyright 2020 © Artgraphics. All rights reserved.

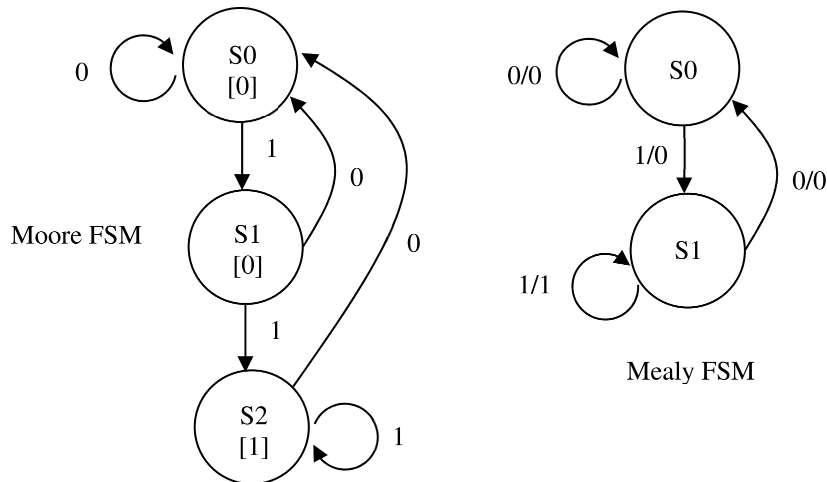


図 17-15　連続した 2 つの 1 を認識する FSM の状態遷移図

この場合には、Moore FSM の S1 と S2 は出力を除くと、全く同じ機能を持っています。そのため、Mealy FSM では、S1 と S2 をマージして状態数を減少させる事ができます。

カウンターに次いで、最も良く知られている FSM は、odd、又は even パリティチェッカーです。odd パリティチェッカーでは、現在までのビット 1 の数が奇数であれば、1 を出力し、偶数であれば、出力は 0 となります。図 17-16 は odd パリティチェッカーを示します。

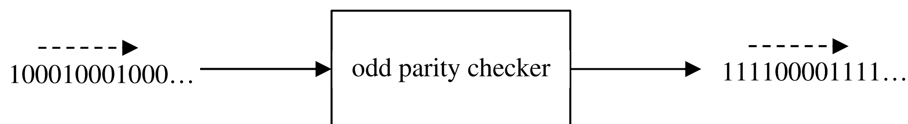


図 17-16　パリティチェッカーのブロックダイアグラム

このパリティチェッカーは、入力と現在の状態から次の状態と出力が決定されるので、FSM になります。この FSM の状態は、Even と Odd の二つの状態から構成されます。以下では、パリティチェッカーに対して Moore FSM と Mealy FSM によるモデリングを解説します。

17.7.4.2　Moore FSM モデリング

Moore タイプの FSM は、図 17-17 に様な構成になります。出力は、レジスタの内容から組み合わせ回路で計算されます。

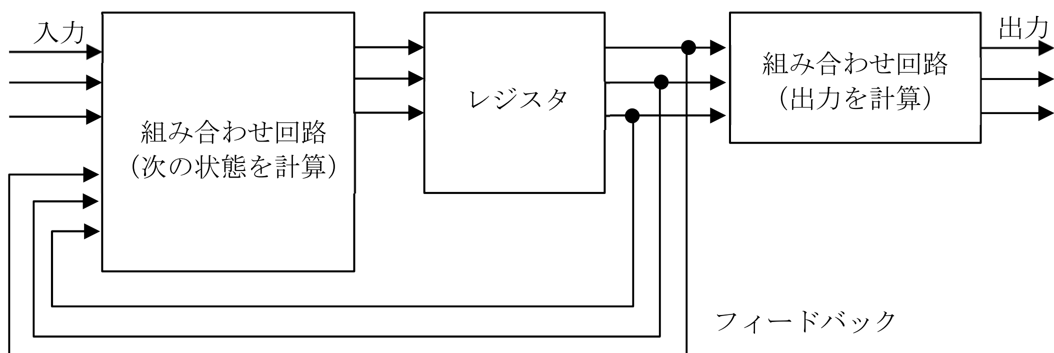


図 17-17　Moore FSM（[9]）

Moore FSM のモデリングは、以下の様な構造を取ります。

Copyright 2020 © Artgraphics. All rights reserved.

21　ファンクショナルカバレッジ

近年の検証手法では、デザインの大規模化に対応するためのテスト法としてランダムスティミュラス生成機能を使用します。ランダムにテストデータを生成して検証する方法は便利で効果的なアプローチですが、実行したテストデータにより検証空間がどれだけカバーされたかを把握しなければ、検証計画を正確に管理する事はできません。テストデータを自動生成する検証手法には、検証に使用されたテストデータを自動的に集計する機能が必要になります。SystemVerfilog のファンクショナルカバレッジはテストデータの自動集計機能を提供します。本章では、ファンクショナルカバレッジの概念と機能を解説します。

21.1　概要

ファンクショナルカバレッジは仕様のどれだけの部分が検査されたかを示す指標です。デザインを検証する意味ではなく、寧ろ、検査者、又は、検査計画の進捗度を示します。ゴールは、勿論、100%のカバレッジです。

ファンクショナルカバレッジは仕様を基にして確認をします。確認を漏れなく、重複が無い様に進める必要があります。例えば、

```
rand bit [2:0]   port;
```

に於いて、ランダム変数 port には乱数が割り当てられます。検査で port が 0 から 7 の全ての値をとればカバレッジは 100%となりますが、一部の値を取らない場合、乱数発生に制約を追加して取り得る値を拡張しなければなりません。或いは、テストする回数を増加しなければなりません。

SystemVerilog では、ファンクショナルカバレッジの仕様をソースコード中に記述する事ができます。しかも、記述されたカバレッジ仕様はシミュレータにより実行されます。カバレッジ仕様を covergroup 文で記述し、covergroup に定義されているビルトインメソッド sample()を呼び出すとカバレッジ計算が行われます。

近年の検証環境におけるテストベンチでは、コレクターが DUT からのレスポンスを収集し、コレクター、又はモニターがカバレッジ計算を行います。図 21-1 は、その様子の一例を示しています。更に詳細な解析をするために、モニターは他の検証コンポーネントに DUT からのレスポンスをトランザクションとして送信します。これらの検証コンポーネントは、SystemVerilog クラスを使用して開発されます。

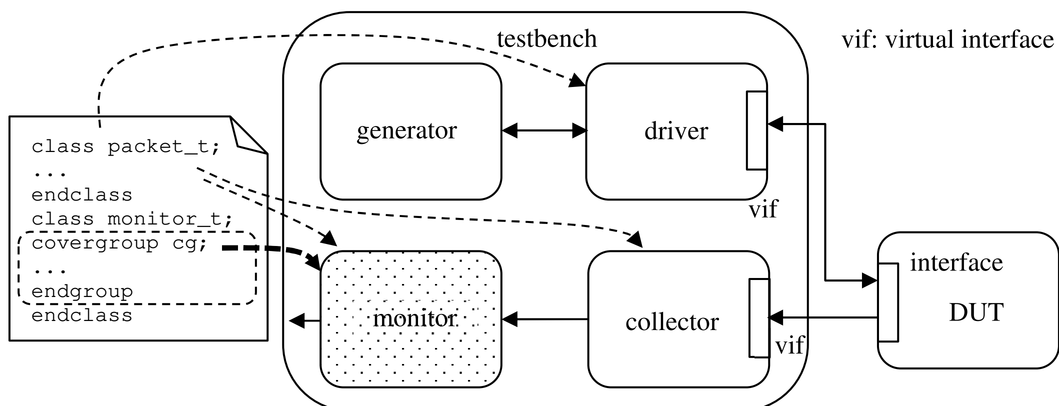


図 21-1　テストベンチに於けるカバレッジ計算の流れ

コレクターが DUT からのレスポンスを収集する時点では、テストに使用された DUT への入力と出力の対が確定しているので、コレクターはそれらの情報をトランザクションとして構成します。このトランザクションにより、実行結果の解析を完全に遂行する事ができます。

Copyright 2020 © Artgraphics. All rights reserved.

例えば、予め計算モデルを準備しておけば、正しい出力か否かを検証する事ができます。コレクターやモニターでは、その実行結果解析の一部分を担います。他の詳細な解析作業は、スコアボードやチェッカーに委ねられます。そのために、モニターはトランザクションを他の検証コンポーネントに送信します。

カバレッジ計算は、比較的簡単な解析処理であるため、しばしばモニターがカバレッジ計算を行います。その場合、モニターの定義には以下の様にカバレッジ定義を含みます。

```
class monitor_t;
...
    covergroup cg with function sample(simple_item item);
        ...
    endgroup

    function void perform_coverage(input simple_item item);
        ...
        cg.sample(item);
    endfunction

endclass
```

この様にカバレッジ定義をすると、sample()メソッドを介してトランザクションで使用されている変数に関するカバレッジ計算を行う事ができます。即ち、DUT で使用された入出力信号に関するカバレッジ計算を行う事ができます。ここで、モニターがトランザクションをコレクターから受信すると、perform_coverage()メソッドが起動する様に設定しておきます。この様にして、ランダムスティミュラスを自動生成して自動検証する環境に自動カバレッジ計算を組み込む事ができます。

21.2 カバレッジモデルの定義

21.2.1 シンタックス

SystemVerilog の covergroup はカバレッジ仕様を定義するための構文です。実際には、クラスと同じ様にデータタイプなので、new コンストラクタを使用してカバーグループのインスタンスを作ります。covergroup のシンタックスは以下の様になります（[1]）。

```
covergroup_declaration ::=
    covergroup covergroup_identifier [ ( [ tf_port_list ] ) ]
        [ coverage_event ] ;
        { coverage_spec_or_option }
    endgroup [ : covergroup_identifier ]

coverage_event ::=
    clocking_event
    | with function sample ( [ tf_port_list ] )
    | @@( block_event_expression )
```

カバーグループには以下の指定を含む事ができます。

- カバーグループ名称（covergroup_identifier）
- カバーグループの外部から取り込む変数名（tf_port_list）
- サンプリングするタイミングを示すカバレッジイベント（coverage_event）
- カバレッジ情報を集計する sample()メソッド
- カバーポイント
- クロスカバレッジ
- カバレッジオプション

Copyright 2020 © Artgraphics. All rights reserved.

の様にして使用します。ここで、s1 と s2 は独立したシーケンスです。シーケンス s1 はシーケンス s2 とは無関係に実行しています。シーケンス s2 において、@(posedge clk) が発生した時、a==1'b1 であれば、1 クロックサイクル後に、s1 はマッチしなければなりません。s1 が何時開始されるかは問題ではありません。

図 22-8 に於いて、シーケンス s1 は時刻 T でマッチすると仮定しています。シーケンス s2 は時刻 T の次のクロッキングイベントで b==1'b1 なのでマッチします。

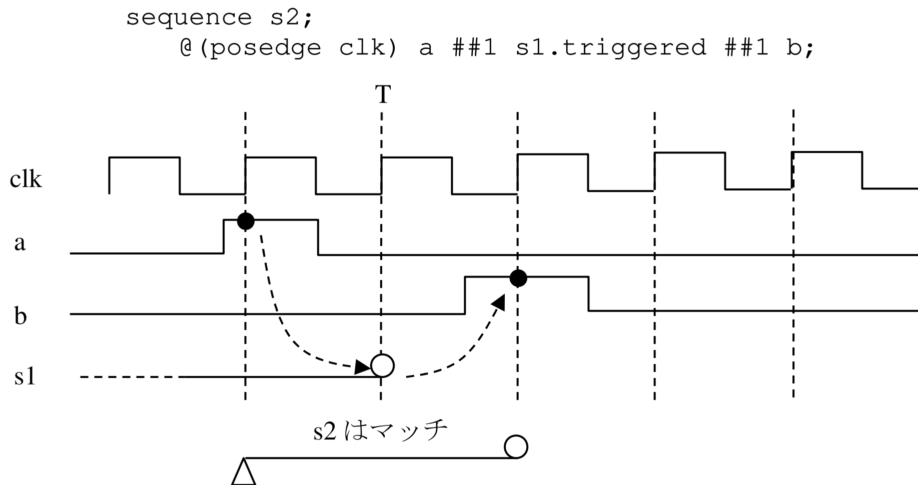


図 22-8　triggered メソッドの使用例

例 22-4　triggered メソッドの使用例

検証すべきプロパーティと独立に実行しているシーケンス unlock がマッチしている状態を参照する例を紹介します。下記のプロパーティ p では、現在のクロッキングイベント時に $rose(check)==1'b1 で、x が真であれば、その後の 1~2 クロックサイクルの間に unlock.triggered が真になる事を要求しています。

```
module test;
bit clk, a, b, c, x, y, check;

   sequence unlock;
      @(posedge clk) a ##1 b ##1 c;
   endsequence

   property p;
      @(posedge clk) $rose(check) |->
              x ##[1:2] unlock.triggered() ##1 y;
   endproperty

   A1: assert property (p)
          else $display("@%3t: FAIL",$time);
   A2: cover property (p)
          $display("@%3t: PASS",$time);

   initial begin
      fork
         begin #20 check = 1; #20 check = 0; end
         begin #20 x = 1; #20 x = 0; end
         begin #80 y = 1; #20 y = 0; end
         begin #20 a = 1; end
```


Copyright 2020 © Artgraphics. All rights reserved.

```
            begin #40 b = 1; end
            begin #60 c = 1; end
        join
    end
    initial repeat( 11 ) #10 clk = ~clk;
endmodule
```

実行結果は以下の様になります。

```
@ 90: PASS
```

関連するアサーションダイアグラムは、図 22-9 の様になります。スレッドの活動は、表 22-6 の様になります。

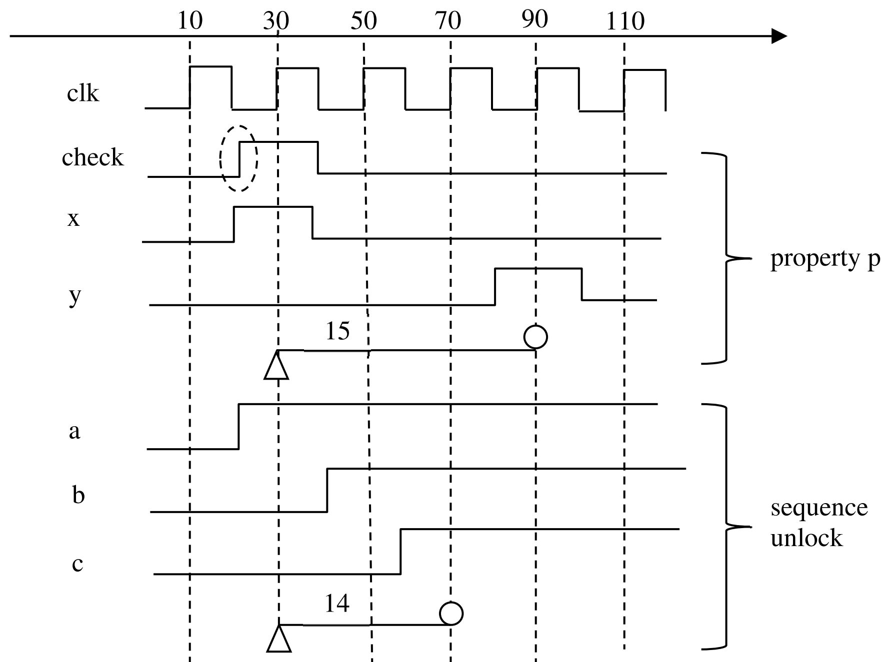


図 22-9　triggered メソッドの使用例

表 22-6　スレッドの活動

$time	スレッド	スレッドの活動
30	14	シーケンス unlock のスレッドが$time==30 で開始します。このスレッドは、$time==70 で pass します。
	15	$time==30 において、$rose(check)==1'b1 で x==1'b1 なので、スレッド 15 は、その後の 1~2 クロックサイクルの間に unlock.triggered が真になるのを待ちます。 $time==70 でスレッド 14 が pass するので、スレッド 15 の unlock.triggered はマッチします。 その直後のクロックサイクルで y==1'b1 なので、スレッド 15 は $time==90 で pass します。

■

22.4.4　便利なサンプル関数

アサーションで良く使用される便利なサンプル関数を表 22-7 にまとめておきます。clocking_event には、@(posedge fclk)の様に指定します。

Copyright 2020 © Artgraphics. All rights reserved.

23.1.3 UVM テストベンチの構成

図 23-3 は UVM のテストベンチの構成を示しています。図において、図の包含関係はクラス、又は、モジュールのインスタンスを持つという関係を意味します。例えば、テストベンチは DUT のインスタンスを保有します。包含関係は、一般的に、階層を意味します。図から明らかな様に、UVM のテストベンチの構成は階層的テストベンチ記述法の構造を反映しています。

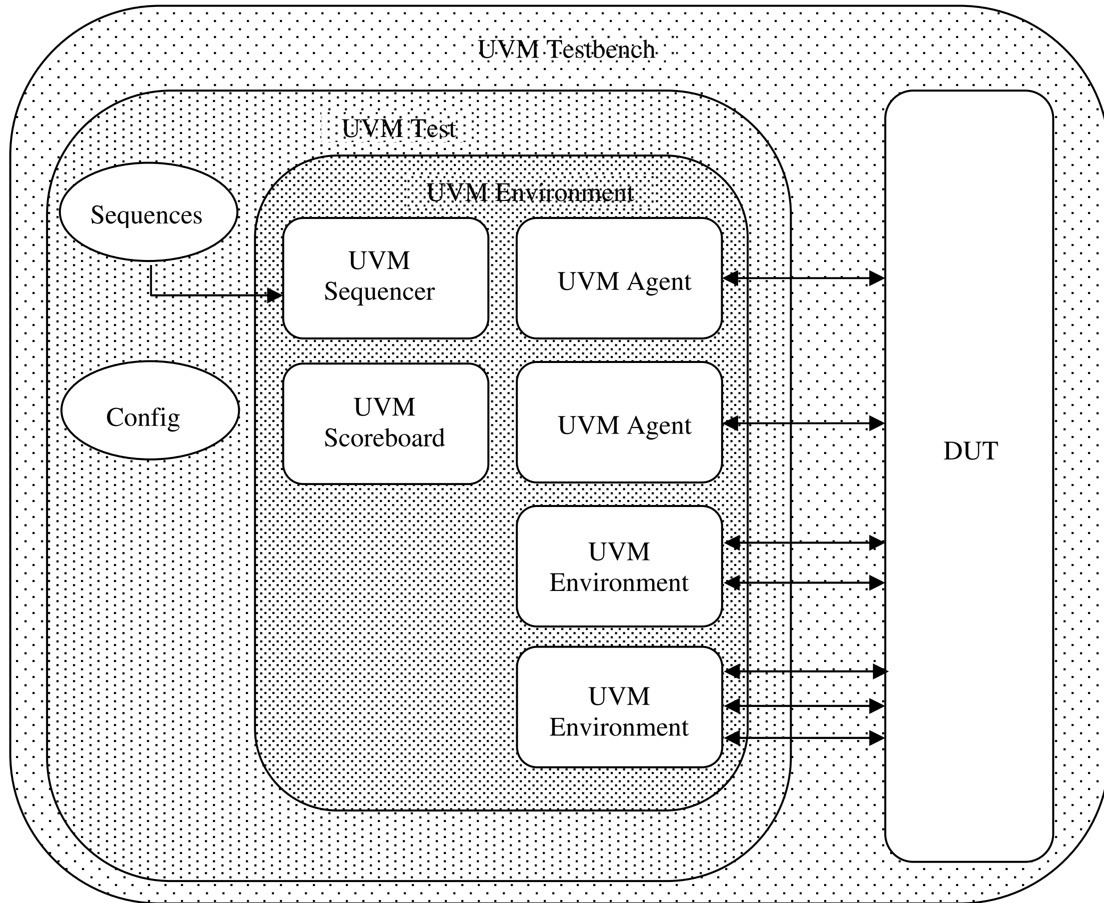


図 23-3 UVM テストベンチの構造[2]

階層を構成する要素の意味を表 23-2 に記します。

表 23-2 階層を構成する UVM の要素

階層を構成する要素	意味
UVM Testbench テストベンチ	これは、所謂、トップレベルのモジュールを指します。このテストベンチは DUT、及び検証機能等をインスタンスとして保有します。
UVM Test テスト	検証用のテストケースのトップレベルのクラスインスタンスを意味します。このインスタンスが該当するテストを実行します。コンフィギュレーション（Config）の設定を変更する事により、各種のテストケースの選別をする事ができます。
UVM Environment エンバイロンメント	階層的に構築した検証コンポーネントです。例えば、トップレベルのエンバイロンメントとしては、SoC、IP（PCIe エンバイロンメント、USB エンバイロンメント、Memory Controller エンバイロンメント）等があります。

Copyright 2020 © Artgraphics. All rights reserved.

UVM Scoreboard スコアボード	スコアボードの主目的は、DUT の動作を確認する事です。即ち、エージェントから取得したDUTのレスポンスが期待される結果と一致するかのチェックを遂行します。
UVM Agent エージェント	基本的な検証コンポーネントで、通常、シーケンサー、ドライバー、モニター、コレクター等のインスタンスで構成されます。エージェントはDUTとvirtualインターフェースを介して接続されます。
UVM Sequencer シーケンサー	トランザクションを生成する制御を司るメインコンポーネントです。シーケンサーはシーケンスを作り、シーケンスを実行する事によりトランザクションを生成します。
Sequences シーケンス	トランザクションを生成するための手順を含むオブジェクトです。オブジェクトであるため、コンポーネント階層には含まれません。シーケンスは複雑なテストシナリオを定義する事ができます。シーケンスはテストに依存するため、エンバイロンメントには含まれていません。
UVM Driver ドライバー	ドライバーはシーケンサーからトランザクションを取得し、シグナルレベルに変換した後、DUT をドライブします。シーケンサーとドライバーの通信にはTLMが使用されます。
UVM Monitor モニター	モニターはDUTからのレスポンスを他の検証コンポーネントにトランザクションとして伝達する役目を持ちます。モニター自身もカバレッジ計算、及びレスポンスに関するチェックも行います。DUT からのレスポンスはシグナルレベルであるため、レスポンスをトランザクションに変換する機能をコレクターとして分離するのが一般的です。

23.1.4　UVMの構成

23.1.4.1　uvm_pkg

UVMの全てのクラスはパッケージuvm_pkgに定義されています。従って、このパッケージをインポートする事により、UVMを使用する事ができる様になります。

23.1.4.2　uvm_object

UVMはSystemVerilogクラスとUVMマクロから構成されます。UVMには多くのクラスが定義されていますが、特別なクラスとして uvm_object が存在します。このクラスはアブストラクトクラスで他の全てのUVMクラスのベースクラスになっています。

uvm_object クラスでは他の全てのクラスに共通する属性、及び手順を宣言しています。具体的な手順の内容はサブクラスで行います。手順としては、例えば、print()及び copy()があります。uvm_object はアブストラクトクラスなので、オブジェクトを作成する事はできませんが、ハンドルを宣言する事はできます。

23.1.4.3　UVMマクロ

マクロには大きく分けて 2 種類あります。一つは、実マクロで、他方は補助マクロです。実マクロは、実行命令に展開されるマクロで比較的分かり易いマクロです。補助マクロは、他の機能への情報を生成するためのマクロで、理解し難いマクロです。例えば、次のマクロは実マクロです。

```
`uvm_info(“INFO”,”Hello world!”,UVM_LOW)
```

このマクロはメッセージをプリントする命令を生成します。この機能は容易に想像する事ができると思います。一方、次の例には補助マクロが使用されています。

```
class simple_item extends uvm_sequence_item;
rand int unsigned addr;
```


Copyright 2020 © Artgraphics. All rights reserved.

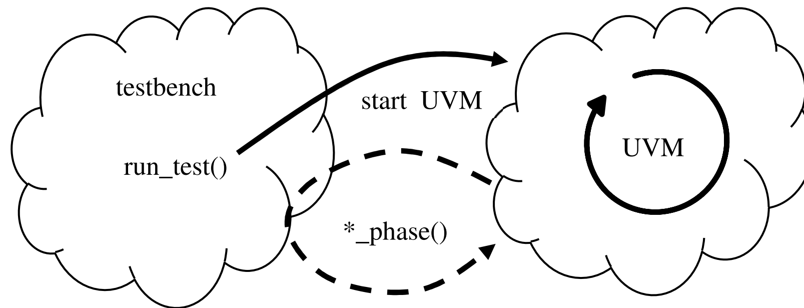


図 23-21　UVM を使用した環境での実行の流れ

UVM コンポーネントが呼び出されるタイミングは予め決定されていてシミュレーションフェーズと呼ばれています。これらのフェーズは virtual タスク、又は、ファンクションとして定義されています。表 23-10 のシミュレーションフェーズは記述されている順に UVM から実行制御を受けます。例えば、run_test()が呼ばれて実行が開始すると、選択されているトップレベルのコンポーネントの build_phase()が呼ばれて階層構造を順に構築します。全てのコンポーネントに対して build_phase() が完了すると、構築された階層構造を基にして、connect_phase()が順に呼ばれて実行します。そして、全てのシミュレーションフェーズが終了すると UVM のシミュレーションの終了となります。

表 23-10　シミュレーションフェーズ

制御順序	フェーズメソッド	機能
1	build_phase ファンクション	コンポーネントの階層を構築するフェーズです。従って、階層のトップから順に呼ばれて行きます。 通常、サブコンポーネントをこのフェーズで作成します。サブコンポーネントを作ると、そのサブコンポーネントの build_phase()が次に呼ばれるので、階層構造が順に構築されます。 また、サブコンポーネントを作る直前に、サブコンポーネントのコンフィギュレーションを変更する事ができます。
2	connect_phase ファンクション	コンポーネント間の接続を完成するフェーズです。例えば、TLM ポートの接続を定義します。或いは、virtual インターフェースの設定等を行います。このフェーズはボトムアップの順序で呼ばれます。 このフェーズが実行する時には、既に決定されているコンフィギュレーションに合わせて TLM 接続をしなければなりません。
3	end_of_elaboration_phase ファンクション	全ての接続が終了するとこのフェーズに制御が移ります。通常は、コンフィギュレーションを確認するためのプリント処理を記述します。
4	start_of_simulation_phase ファンクション	シミュレーションが開始する直前にこのフェーズが呼ばれます。初期化処理等を行う事ができます。
5	run_phase タスク	シミュレーションを行うためのフェーズです。実際にシミュレーションを進行させるためには、テストベンチを構成する少なくとも一つのコンポーネントが raise_objection() と

Copyright 2020 © Artgraphics. All rights reserved.

		drop_objection()を呼び出さなければなりません。 全ての検証コンポーネントが、このフェーズを定義する必要はありません。通常、DUTと関わりのあるコンポーネントに対してrun_phase()を定義します。例えば、ドライバーやコレクターにはrun_phase()を定義する必要があります。
6	extract_phase ファンクション	シミュレーションが終了すると、このフェーズに制御が移ります。シミュレーション結果を抽出するための処理を記述する事ができます。
7	check_phase ファンクション	抽出したシミュレーション結果をチェックするための処理を記述します。
8	report_phase ファンクション	シミュレーション結果のレポートを出力する処理を記述します。

ユーザは必要なフェーズだけを記述すれば良い事になっています。run_phase() は時間を消費するシミュレーションフェーズであるため、タスクとして定義されていますが、その他のフェーズはファンクションとして定義されています。

図23-22はドライバーの定義とシミュレーションフェーズがUVMから制御を受ける様子を示しています。

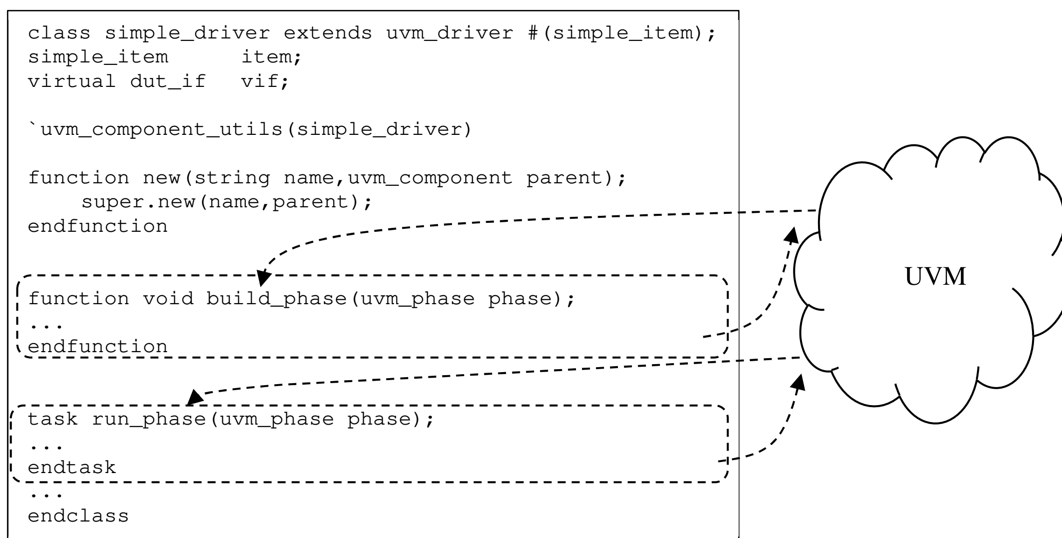


図23-22　UVMによるシミュレーション制御

以下はbuild_phase()の記述例を示しています。

```
function void simple_driver::build_phase(uvm_phase phase);
   super.build_phase(phase);
   if( !uvm_config_db#(virtual dut_if)::get(this,"","vif",vif) )
      `uvm_fatal("NO-VIF",{"VIF must be set for:",
          get_full_name(),",vif"})
endfunction
```

シミュレーションフェーズは virtual メソッドであるため、一定のルールがあります。以下のルールは重要です。

Copyright 2020 © Artgraphics. All rights reserved.