

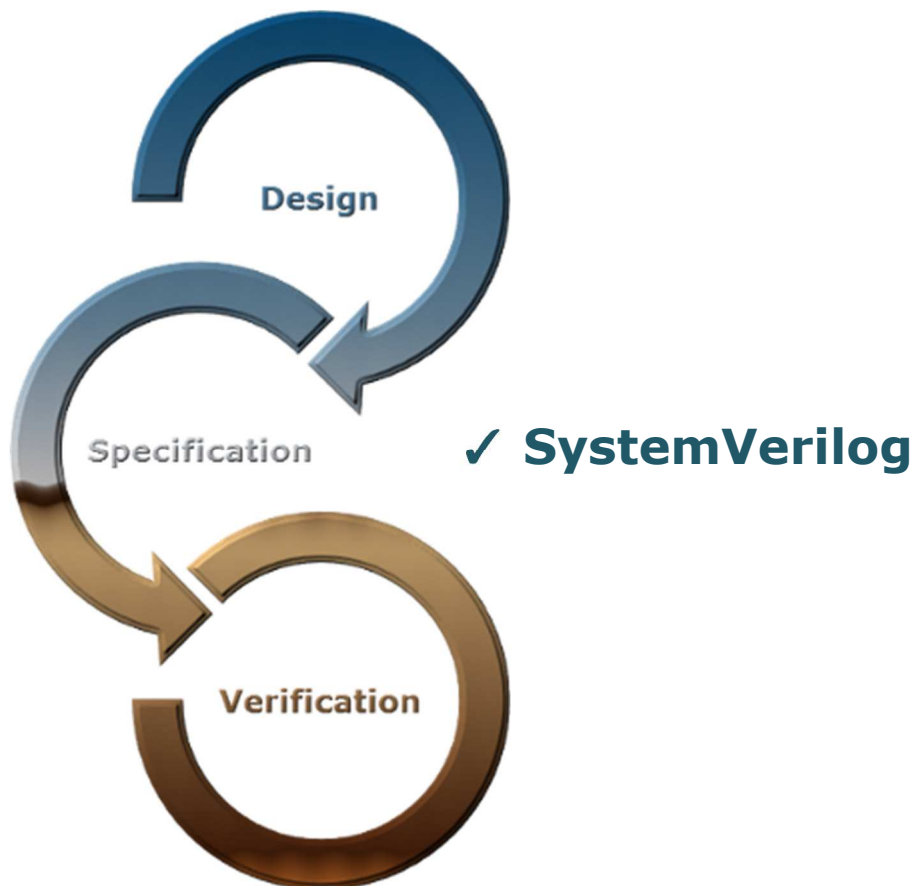
実践 SystemVerilog 入門

Document Identification Number: ARTG-TD-001-2019

Document Revision: 1.1, 2020.03.16

アートグラフィックス

篠塚一也



実践 SystemVerilog 入門

© 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

A Practical Guide to Learning SystemVerilog

© 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らると思えます。

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

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よう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

変更履歴

日付	Revision	変更点
2020.02.16	1.0	初版。
2020.03.16	1.1	① FSM に関する例を変更。

目次

1	概要	1
1.1	SYSTEMVERILOG の歴史	1
1.2	SYSTEMVERILOG 概要	1
1.2.1	言語としての SystemVerilog	1
1.2.2	設計言語としての SystemVerilog	2
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	11
2.4	インターフェース	12
2.5	チェッカー	13
2.6	パッケージ	14
2.7	ゲートとスイッチレベルのモデリング	15
2.8	PRIMITIVE	17
2.9	CONFIGURATION	17
2.10	コンパイルユニット	17
2.11	`TIMESCALE コンパイラダイレクティブ	18
2.12	VERILOG から SYSTEMVERILOG への移行	20
2.12.1	背景と動機	20
2.12.2	Verilog から SystemVerilog への変換	20
2.12.2.1	組み合わせ回路	20
2.12.2.2	シーケンシャル回路	22
3	データタイプ	23
3.1	データタイプとデータオブジェクト	23
3.2	LOGIC 型	24
3.3	ネット型	24
3.4	変数	26
3.5	単数系と集合系	28
3.6	ネットと変数	28
3.7	4-STATE 型	29
3.8	2-STATE 型	30
3.9	INTEGRAL データタイプ	31
3.10	REAL、SHORTREAL と REALTIME	32
3.11	VOID 型	32
3.12	CHANDLE 型	32
3.13	クラス	32
3.14	STRING データタイプ	32
3.15	イベントデータタイプ	35
3.16	TYPDEF	36
3.17	ENUM データタイプ	37
3.18	定数	41
3.19	CAST オペレータ	44
3.20	\$CAST ダイナミック型変換	45

3.21	便利な初期値設定.....	46
3.21.1	リテラルの拡張.....	46
3.21.2	インデックス指定.....	47
3.22	リファレンスポインター.....	47
4	メンバーで構成されるデータタイプ.....	49
4.1	ストラクチャ.....	49
4.1.1	Packed ストラクチャ.....	51
4.1.2	ストラクチャへの値の設定.....	52
4.2	ユニオン.....	52
4.2.1	Packed ユニオン.....	54
4.2.2	タグ付きユニオン.....	54
4.3	PACKED アレイと UNPACKED アレイ.....	55
4.3.1	Packed アレイ.....	55
4.3.2	Unpacked アレイ.....	56
4.3.3	アレイの操作.....	57
4.3.4	packed アレイのアクセス.....	59
4.4	ダイナミックアレイ.....	59
4.4.1	ダイナミックアレイのメソッド.....	60
4.4.2	アレイのコピー.....	62
4.5	ASSOCIATIVE アレイ.....	63
4.5.1	Associative アレイの概要.....	63
4.5.2	Associative アレイの要素の登録.....	64
4.5.3	Associative アレイの関数.....	65
4.5.4	Associative アレイリテラル.....	66
4.6	キュー.....	67
4.6.1	キューの概要.....	67
4.6.2	キューの操作.....	68
4.6.3	キューを操作する関数.....	70
4.7	アレイ情報取得関数.....	72
4.8	アレイ操作関数.....	73
4.8.1	アレイ検索関数.....	73
4.8.2	アレイ要素の順序を操作する関数.....	75
4.8.3	アレイを計算する関数.....	76
4.9	アレイの走査法.....	77
5	クラス.....	79
5.1	クラスの概要.....	79
5.2	シンタックス.....	80
5.3	クラスオブジェクト (クラスインスタンス).....	82
5.4	クラスプロパティ及びメソッドへのアクセス.....	82
5.5	コンストラクタ.....	83
5.6	タイプ指定のコンストラクタ呼び出し.....	83
5.7	STATIC クラスプロパティ.....	84
5.8	STATIC クラスメソッド.....	86
5.9	THIS ハンドル.....	86
5.10	ハンドルのアレイ.....	87
5.11	クラスのコピー.....	88
5.12	クラスインヘリタンスとサブクラス.....	89
5.13	\$CAST.....	92
5.14	VIRTUAL メソッド.....	93
5.15	アブストラクトクラスとピュアバーチャルメソッド.....	95
5.16	クラススコープオペレータ.....	96

5.17	メンバーへのアクセス制限	98
5.18	メソッドをクラスの外に記述する方法	99
5.19	パラメータによる汎用クラスの定義	100
5.19.1	概要	100
5.19.2	パラメータによる汎用クラスの実装例	100
5.19.3	パラメータによる汎用クラスの開発手順	102
5.19.4	Quicksort	103
5.20	クラスのフォワード宣言	105
5.21	インターフェースクラス	106
5.21.1	概要	106
5.21.2	機能	107
5.22	ガーベッジコレクション	109
5.22.1	概要	109
5.22.2	automatic 変数	109
5.22.3	static 変数	110
6	プロセス	111
6.1	シミュレーションプロシージャ	111
6.1.1	initial プロシージャ	112
6.1.2	always プロシージャ	113
6.1.2.1	組み合わせ回路とセンシティブティリスト	113
6.1.2.2	シーケンシャル回路と always プロシージャ	113
6.1.2.3	always_comb プロシージャ	115
6.1.2.4	always_latch プロシージャ	116
6.1.2.5	always_ff プロシージャ	117
6.1.3	final プロシージャ	119
6.2	ブロック文	119
6.2.1	begin-end ブロック	120
6.2.2	fork-join ブロック	121
6.2.3	ブロック名	125
6.2.4	fork ブロックの効果的利用	125
6.3	タイミングによる実行制御	126
6.3.1	タイミングによる実行制御の概要	126
6.3.2	ディレーによる制御	127
6.3.3	エッジセンシティブイベント制御	127
6.3.3.1	機能	127
6.3.3.2	センシティブティリスト指定の注意点	129
6.3.4	代入内タイミング制御	131
6.3.5	レベルセンシティブイベント制御	134
6.3.6	イベント制御と解除	134
6.3.7	シーケンスによるイベント制御	135
6.4	プロセス制御	136
6.4.1	wait fork 文	136
6.4.2	disable fork 文	138
6.4.3	wait_order 文	138
6.5	プロセスと RNG	139
6.6	ユーザ固有のプロセス制御	141
7	代入文	144
7.1	連続代入文	144
7.2	ビヘイビア代入文	146
7.2.1	ブロッキング代入文	146
7.2.2	ノンブロッキング代入文	147
7.2.2.1	機能	147

7.2.2.2	代入文の効果の確認.....	148
7.2.2.3	左辺に関する制限.....	150
7.3	パターン指定による代入.....	150
7.4	アレイパターン指定代入文.....	151
7.5	ストラクチャパターン指定代入文.....	152
7.6	UNPACKED アレイの初期化.....	153
8	オペレータと式.....	155
8.1	オペレータ.....	155
8.1.1	代入オペレータ.....	156
8.1.2	インクリメント及びデクリメントオペレータ.....	156
8.1.3	算術演算子.....	157
8.1.4	比較オペレータ.....	158
8.1.5	ワイルドカード比較オペレータ.....	159
8.1.6	logical オペレータ.....	160
8.1.7	bitwise オペレータ.....	161
8.1.8	計算オペレータ.....	163
8.1.9	shift オペレータ.....	164
8.1.10	conditional オペレータ.....	165
8.1.11	結合オペレータ.....	166
8.1.12	inside オペレータ.....	167
8.1.13	ビットストリームオペレータ.....	168
8.2	オペランド.....	170
8.2.1	パートセレクト.....	170
8.2.2	unpacked アレイ.....	171
8.3	タグ付きメンバーの操作.....	172
9	実行文.....	173
9.1	IF 文.....	173
9.1.1	全ての条件を列挙.....	173
9.1.2	unique-if、unique0-if 文.....	173
9.1.3	priority-if 文.....	174
9.2	CASE 文.....	175
9.2.1	unique-case、unique0-case 文.....	176
9.2.2	priority-case 文.....	177
9.2.3	inside オペレータと if 文及び case 文.....	177
9.2.3.1	if 文と inside オペレータ.....	177
9.2.3.2	case 文と inside オペレータ.....	177
9.2.1	casez と casex.....	178
9.3	ループ文.....	179
9.3.1	for 文.....	180
9.3.2	repeat 文.....	180
9.3.3	foreach 文.....	181
9.3.4	while 文.....	183
9.3.5	do-while 文.....	184
9.3.6	forever 文.....	185
9.4	RETURN 文.....	185
9.5	BREAK 文.....	186
9.6	CONTINUE 文.....	187
10	タスクとファンクション.....	189
10.1	タスク.....	189
10.1.1	ポートリスト.....	189

10.1.2	タスク内の記述	189
10.2	ファンクション	190
10.2.1	ファンクションの制限	190
10.2.2	ポートリスト	190
10.2.3	ファンクション内の記述	191
10.2.4	ファンクション内でタイミング制御を行う方法	192
10.3	引数に標準値を指定する方法	193
10.4	値を戻すファンクションの使用	194
10.5	値を戻さないファンクションの使用	194
10.6	再帰呼び出し	195
10.7	クラスのメソッドと再帰呼び出し	196
10.8	メソッド内での変数の初期化	196
10.9	引数としてのアレイ	199
10.10	インポートとエクスポート	199
11	クロッキングブロック	201
11.1	最も簡単なクロッキングブロック	201
11.2	クロッキングキュー	202
11.3	クロッキングイベントと OBSERVED 領域	204
11.4	サイクルディレイ	205
12	プロセス間の同期と交信	207
12.1	セマフォ	207
12.2	メールボックス	210
12.3	パラメータ化したメールボックス	214
12.4	名称付きイベント	215
12.4.1	概要	215
12.4.2	triggered	217
12.4.3	引数としてのイベントオブジェクト	219
12.4.4	イベント資源の解放	220
12.4.5	イベントの比較	221
12.4.6	イベントの別名	221
13	チェッカー	222
13.1	概要	222
13.2	チェッカーインスタンス	222
13.3	自由変数	224
14	プログラム	226
14.1	シンタックス	226
14.2	プログラムの特徴	227
14.3	プログラムの制御	229
14.4	シミュレーションの終了	229
14.5	テストベンチ	230
15	インターフェース	232
15.1	シンタックス	232
15.2	インターフェースの機能概要	233
15.3	ジェネリックインターフェースに依る接続	234
15.4	MODPORT	234
15.5	パラメータ化したインターフェース	236
15.6	VIRTUAL インターフェース	237
15.7	インターフェース使用例	238

15.7.1	概要	238
15.7.2	simple_if.....	239
15.7.3	driver.....	239
15.7.4	collector.....	240
15.7.5	dut.....	240
15.7.6	top.....	240
15.7.7	実行結果.....	241
16	パッケージ	242
16.1	シンタックス	242
16.2	パッケージの定義法	243
16.3	パッケージの使用法	243
16.4	STD パッケージ.....	246
17	モジュール	250
17.1	概要.....	250
17.2	モジュールの定義.....	251
17.3	ポートリスト	253
17.3.1	Verilog スタイルと SystemVerilog スタイル.....	253
17.3.2	ポートの方向に関するルール	253
17.4	パラメータ化したモジュール.....	254
17.5	トップレベルモジュール	256
17.6	モジュールインスタンス	256
17.7	モジュール記述例	257
17.7.1	一般的な記述.....	257
17.7.2	組み合わせ回路	259
17.7.2.1	組み合わせ回路の検証	259
17.7.2.2	組み合わせ回路の記述ルール.....	261
17.7.2.3	ALU	261
17.7.2.4	コンパレータ.....	263
17.7.2.5	デコーダー	264
17.7.2.6	エンコーダー.....	266
17.7.2.7	Gray コード	267
17.7.2.8	multiplexer.....	269
17.7.2.9	バレルシフタ.....	271
17.7.2.10	ファンクションユニット	272
17.7.2.11	符号付き整数の加減算	274
17.7.3	シーケンシャル回路	276
17.7.3.1	シーケンシャル回路の検証	277
17.7.3.2	シーケンシャル回路の検証手順.....	278
17.7.3.3	シーケンシャル回路の記述ルール.....	278
17.7.3.4	バイナリーカウンタ	279
17.7.3.5	ラッチ	281
17.7.3.6	JK-フリップフロップ	283
17.7.3.7	データシフタ.....	284
17.7.3.8	ユニバーサルシフトレジスタ	285
17.7.3.9	Johnson カウンタ	287
17.7.3.10	Gray カウンタ	289
17.7.3.11	リングカウンタ.....	290
17.7.3.12	Gated clock の記述例.....	292
17.7.3.13	インターフェースを使用するモジュール記述.....	294
17.7.4	FSM.....	296
17.7.4.1	概要.....	296
17.7.4.2	Moore FSM モデリング	297
17.7.4.3	Mealy FSM モデリング	299
17.7.5	ref ポートの使用	302

17.7.6	inout ポートを持つシーケンシャル回路	302
17.8	未定義モジュールの宣言	304
17.9	階層名称	305
18	ジェネレート	307
18.1	概要	307
18.2	使用法	308
19	システムタスクとシステム関数	311
19.1	DISPLAY 及び WRITE タスク	311
19.2	\$SFORMAT と \$SFORMATF	313
19.3	モニタリング	313
19.4	シミュレーション時間取得関数	314
19.5	\$PRINTTIMESCALE	315
19.6	値変換	316
19.7	情報取得関数	316
19.8	ビット VECTOR システム関数	319
19.9	サンプル値を参照するための関数	320
19.10	エラー処理タスク	322
19.11	確率分布関数	323
19.12	シミュレーション制御	324
19.13	その他のシステムタスク及びシステム関数	324
19.14	コマンドラインの操作	325
19.15	VCD ファイル	326
19.15.1	VCD ファイルの指定	327
19.15.2	VCD ファイルへの記録	327
19.15.3	VCD ファイルへの記録の一時的停止と再開	327
19.15.4	VCD ファイル作成例	328
20	制約によるランダムステイミュラスの生成	329
20.1	概要	329
20.2	ランダム変数	330
20.2.1	ランダム変数の概要	330
20.2.2	rand 修飾子	331
20.2.3	randc 修飾子	331
20.2.4	rand と randc の使用例	331
20.3	制約	332
20.3.1	inside オペレータ	333
20.3.2	dist オペレータ	335
20.3.3	unique オペレータ	338
20.3.4	implication オペレータ	340
20.3.5	if-else 制約	341
20.3.6	foreach 制約	343
20.3.7	乱数決定順序	344
20.4	乱数発生関数	346
20.5	実行時に制約を定義する方法	348
20.6	ランダム変数の制御	349
20.7	制約の制御	350
20.8	RANDOMIZE()関数によるランダム変数の制御	351
20.9	条件の否定	352
20.10	ストラクチャ	353
20.11	キューに乱数を発生	354
20.12	チェッカーとしての制約	355

20.13	制約をテストケース毎に指定する方法	357
20.14	制約をクラス外部に定義する方法	358
20.15	STD::RANDOMIZE0	359
20.16	システム関数	360
20.17	RANDCASE	361
21	ファンクショナルカバレッジ	363
21.1	概要	363
21.2	カバレッジモデルの定義	364
21.2.1	シンタックス	364
21.2.2	カバーグループへの引数	365
21.2.3	サンプリングのタイミング指定	365
21.2.4	サンプリング関数	365
21.2.5	クラス外でのカバレッジ定義	366
21.2.6	カバレッジレポート	367
21.2.7	簡単な使用例	367
21.3	カバーポイントの定義	368
21.3.1	カバレッジビンの定義	369
21.3.1.1	固定数のビン	369
21.3.1.2	それぞれの値にビンを確保する方法	369
21.3.1.3	唯一つのビン	370
21.3.1.4	auto ビン	370
21.3.1.5	default ビン	371
21.3.1.6	illegal_bins	371
21.3.1.7	ignore_bins	372
21.3.2	カバーポイントの使用例	372
21.3.3	信号値の遷移	379
21.3.4	式のカバレッジ	381
21.3.5	制約とカバレッジ	383
21.4	クロスカバレッジ	384
21.5	自動カバレッジ収集	386
21.5.1	カバレッジ計算と検証コンポーネント	386
21.5.2	カバーグループの定義	387
21.5.3	カバレッジのサンプリング	387
21.5.4	自動カバレッジ収集例	387
21.5.4.1	simple_if	388
21.5.4.2	simple_item	388
21.5.4.3	simple_component	389
21.5.4.4	simple_driver	389
21.5.4.5	simple_generator	390
21.5.4.6	simple_collector	390
21.5.4.7	simple_monitor	391
21.5.4.8	dut	391
21.5.4.9	top	391
21.5.4.10	自動カバレッジ収集のまとめ	392
22	アサーション	393
22.1	概要	393
22.1.1	アサーションとは何か?	393
22.1.2	アサーションの種類	394
22.1.3	サンプリング	394
22.1.4	アサーションクロック	394
22.1.5	アサーションの式	395
22.2	即時実行型アサーション	395
22.3	並列型アサーション	396

22.3.1	概要	396
22.3.2	assert 文.....	397
22.3.3	assume 文	397
22.3.4	cover 文.....	397
22.3.5	restrict 文	398
22.4	IMPLICATION オペレータ	398
22.4.1	機能	398
22.4.2	vacuous pass の抑止.....	399
22.5	アサーションスレッドダイアグラム.....	400
22.6	シーケンス.....	400
22.6.1	シーケンスの定義.....	400
22.6.2	シーケンス式の結合	401
22.6.3	クロッキングイベント	402
22.6.4	レベルセンシティブ とエッジセンシティブ	402
22.6.5	triggered()メソッド.....	403
22.6.6	シーケンスの操作.....	404
22.6.6.1	##m	405
22.6.6.2	##[m:n]	406
22.6.6.3	[*m].....	407
22.6.6.4	[*m:n].....	409
22.6.6.5	[=m]	410
22.6.6.6	[->m]	411
22.6.6.7	sig throughout seq	413
22.6.6.8	seq1 within seq2.....	414
22.6.6.9	seq1 and seq2.....	416
22.6.6.10	seq1 or seq2.....	417
22.6.6.11	seq1 intersect seq2.....	418
22.7	プロパティ (PROPERTIES)	420
22.7.1	概要	420
22.7.2	シーケンスとプロパティ	421
22.8	プロパティオペレータ	421
22.8.1.1	##	423
22.8.1.2	s_always [constant_range]	424
22.8.1.3	until	425
22.9	マルチクロック	427
23	UVM.....	429
23.1	概要.....	429
23.1.1	UVM とは何か?	429
23.1.2	検証技術のトレンド.....	429
23.1.3	UVM テストベンチの構成.....	431
23.1.4	UVM の構成.....	432
23.1.4.1	uvm_pkg	432
23.1.4.2	uvm_object	432
23.1.4.3	UVM マクロ.....	432
23.1.5	ソースコードの準備	433
23.1.5.1	インクルード.....	433
23.1.5.2	uvm_pkg のインポート.....	433
23.2	TLM.....	434
23.2.1	概要	434
23.2.2	トランザクション.....	434
23.2.3	UVM コンポーネント間の通信.....	435
23.2.3.1	概要	435
23.2.3.2	put 操作.....	436
23.2.3.3	get 操作	440
23.2.3.4	uvm_tlm_fifo	444

23.2.3.5	analysis-ports と analysis-exports.....	447
23.3	代表的な UVM クラス.....	451
23.3.1	トランザクション関連の UVM クラス.....	451
23.3.2	メソドロジークラス.....	452
23.4	VIRTUAL インターフェース.....	452
23.5	UVM シミュレーション制御.....	453
23.5.1	シミュレーションフェーズ.....	453
23.5.2	シミュレーションフェーズと super.method 0.....	455
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.....	456
23.5.4	コンポーネント階層の情報取得関数.....	457
23.5.5	シミュレーションの進行.....	457
23.5.5.1	raise_objection()と drop_objection().....	457
23.5.5.2	raise_objection()と drop_objection()の使用例.....	458
23.5.6	run_test.....	460
23.5.6.1	概要.....	460
23.5.6.2	使用法.....	460
23.5.6.3	run_test()と\$finish.....	462
23.6	UVM クラスライブラリーの基礎.....	464
23.6.1	uvm_object と uvm_component.....	464
23.6.2	コンストラクタ.....	464
23.6.3	フィールドマクロ.....	465
23.6.3.1	トランザクション.....	465
23.6.3.2	メソドロジークラス.....	466
23.6.4	ファクトリ.....	466
23.6.5	コンフィギュレーションの設定変更.....	468
23.6.6	メッセージ機能.....	469
23.6.7	UVM プリンター.....	470
23.7	検証要素の定義.....	472
23.7.1	トランザクション.....	472
23.7.2	ドライバー.....	473
23.7.2.1	概要.....	473
23.7.2.2	ドライバー記述法.....	474
23.7.3	シーケンス.....	476
23.7.3.1	概要.....	476
23.7.3.2	シーケンスの定義手順.....	476
23.7.3.3	body()タスク.....	477
23.7.3.4	`uvm_do マクロと`uvm_do_with マクロ.....	477
23.7.3.5	raise_objection()と drop_objection().....	477
23.7.3.6	pre_body()と post_body().....	478
23.7.3.7	シーケンス定義例.....	479
23.7.4	シーケンサー.....	482
23.7.4.1	シーケンサーの定義.....	482
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.....	483
23.7.4.3	シーケンスの開始.....	484
23.7.5	モニターとコレクター.....	484
23.7.5.1	コレクターの定義.....	484
23.7.5.2	モニターの定義.....	485
23.7.6	エージェント.....	486
23.7.7	エンバイロメント.....	487
23.7.8	テストベンチの作成.....	489
23.7.8.1	トップレベルの Environment の作成.....	489
23.7.8.2	ベーステスト.....	490
23.7.8.3	テスト.....	490
23.7.8.4	テストの選択.....	490
23.8	UVM による検証環境構築例.....	491

23.8.1	検証環境の概要	491
23.8.2	pkg_definitions.sv ファイル	493
23.8.3	alu_if.....	493
23.8.4	alu_item	493
23.8.5	alu_driver.....	493
23.8.6	alu_sequencer	494
23.8.7	alu_sequence_base.....	495
23.8.8	alu_test_seq1	496
23.8.9	alu_test_seq2	496
23.8.10	alu_collector	497
23.8.11	alu_monitor	498
23.8.12	alu_agent.....	498
23.8.13	alu_env	499
23.8.14	alu_test_base.....	500
23.8.15	alu_test1	500
23.8.16	alu_test2.....	501
23.8.17	dut.....	501
23.8.18	top	501
23.8.19	テストの実行.....	502
23.8.19.1	+UVM_TESTNAME=alu_test1	502
23.8.19.2	+UVM_TESTNAME=alu_test2.....	503
24	コンパイラダイレクティブ	504
24.1	`INCLUDE 文.....	504
24.2	`DEFINE 文	504
24.2.1	定数を定義する場合	504
24.2.2	接頭辞及び接尾辞を持つ名称の創成.....	505
24.3	文字列内のパラメータ展開	506
24.4	`ENDIF 文.....	506
24.5	`__FILE__、`__LINE__	506
25	シミュレーション実行モデル.....	509
25.1	スケジューリング領域.....	509
25.2	#0 デイレーの効果	510
25.2.1	スケジューリングの調節.....	510
25.2.2	タイミングの調節.....	511
26	DPI	513
26.1	概要.....	513
26.2	IMPORT と EXPORT.....	513
26.2.1	import.....	514
26.2.2	export.....	515
26.3	C LAYER	516
26.3.1	概要	516
26.3.2	4-state 型	517
26.3.3	open アレイ.....	518
27	補足.....	520
27.1	SYSTEMVERILOG 全般	520
27.2	UVM 全般	523
28	参考文献.....	526

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、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ようになります。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。

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

	定めます。アレイの領域も実行時に割り当てる事ができません。
--	-------------------------------

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。

例 3-3 ネット型の宣言例

以下の宣言において、何れのネットも logic 型になります。

```

wire          n1;
wire logic    n2;
wire [15:0]  n3;

```

5 クラス

クラスは、実行時にコンフィギュレーションを構築または変更する事を可能にするため、検証環境の再利用性を促進します。近年では、クラスをベースにした検証環境構築法が主流になりつつあります。本章では、クラスを検証作業に適用するための実践知識を養います。クラスはデータタイプですが、クラスが RTL デザインに使用される事はありません。理由は、クラスは論理合成可能ではないからです。

クラスは、トランザクションの様なデータを表現するだけでなく、データを処理するためのアルゴリズムを記述する手段としても使用されます。クラスを定義すると、クラスのインスタンスを作ることができます。そして、クラスインスタンスの階層構造を利用する事により、モジュールによるデザイン階層と同じ様に複雑な検証環境を実現することができます。検証時の生産性向上をもたらす手段として、クラスは重要な役割を果たします。

5.1 クラスの概要

クラスはデータタイプの一つで、データとデータを操作するためのサブルーティン（タスク、ファンクション）から構成されます。クラスでは、データをプロパティ、サブルーティンをメソッドとも呼びます。クラスには `new` と呼ばれる特別なメソッドがあり、コンストラクタと呼ばれます。コンストラクタはクラスのインスタンス（つまり、オブジェクト）を作ります。SystemVerilog のクラス概念は C++ のそれより Java のクラスに似ています。

例 5-1 クラスの例

この例では、簡単なトランザクションを `simple_item` として定義しています。このクラスにはメソッドとして、`new()`、`get_name()`、及び `print()` が定義されています。

```
class simple_item;
parameter ADDR_WIDTH = 32;
parameter DATA_WIDTH = 32;
typedef logic[ADDR_WIDTH-1:0] addr_t;
typedef logic[DATA_WIDTH-1:0] data_t;

    static int        counter;           // static
    local string     name;             // local
    addr_t           addr;
    data_t           data;

function new(string name,addr_t a=100,data_t d=0);
    this.name = name;
    addr = a;
    data = d;
    counter++;
endfunction

function string get_name();
    get_name = name;
endfunction

    virtual function void print;       // virtual
    $display("name=%s",name);
    $display("addr=%h",addr);
    $display("data=%h",data);
endfunction

endclass
```

表 5-1 は、このクラスの解説を示しています。

表 5-1 クラスの代表的なメンバー

メンバー	属性	意味
new	コンストラクタ	コンストラクタの引数には標準値が与えられています。コンストラクタを呼ぶ際、インスタンス名だけが必須のパラメータです。
counter	static	このメンバーはクラスのインスタンス数を示します。static であるため、このクラスの全てのインスタンスに対して唯一つの counter が確保されます。データタイプは int なので、counter の初期値は 0 となっています。クラスのインスタンスを作る時には、new コンストラクタが必ず呼ばれます。従って、new() メソッドで counter の更新をします。
name	local	メンバーは local として宣言されているため、クラスの外から直接参照する事はできません。クラスの外から name を参照するためには、ファンクション get_name() を使用しなければなりません。
addr data	パブリック	属性を指定していないので、パブリックとなります。即ち、クラス外からも直接参照する事が出来ます。
print	virtual	この関数は virtual として定義されています。このクラスから継承したクラスでこの関数を書き換える事ができます。

5.2 シンタックス

以下は、クラス全体を示すシンタックスです ([1])。前記の例を参照しながらシンタックスを確認して下さい。

```
class_declaration ::=
  [virtual] class [ lifetime ]
  class_identifier [ parameter_port_list ]
  [ extends class_type [ ( list_of_arguments ) ] ]
  [ implements interface_class_type { , interface_class_type } ] ;
  { class_item }
  endclass [ : class_identifier]
```

複雑なシンタックスであるため、詳細なシンタックスの解説を避け、実践に必要な使い方を順次説明します。ここでは、概略に留めておきます。

- ① クラスの定義はキーワード `class` で始まり、キーワード `endclass` で終了します。 `endclass` キーワードの後に、コロンを挟んでクラス名称を添える事もできます。この名称は、コメントとして役立ちます。アブストラクトクラスを定義する際には、キーワード `class` の前に `virtual` の指定が必要です。
- ② キーワード `class` の後には、 `lifetime` の指定をすることができます。ここで、 `lifetime` とは `static` 又は `automatic` の何れかを指定します。省略すると `automatic` が仮定されます。即ち、クラス内のプロパティ、及びメソッドは原則として `automatic` です。明示的に `static` と宣言しない限り `automatic` になります。
- ③ `class_identifier` はクラス名称で必須な項目です。クラスを汎用的にするためには、 `parameter_port_list` でパラメータを指定します。
- ④ `extends` キーワードは、クラスインヘリタンスを意味します。即ち、既存のクラスを利用して新しいクラスを定義する場合に、 `extends` を指定します。既存のクラスにパラメータが必要な場合、 `(list_of_arguments)` に於いてパラメータを指定します。
- ⑤ `interface` クラスに実装を追加する場合に、キーワード `implements` を指定します。



12.2 メールボックス

メールボックスは **producer** と **consumer** をもつ **FIFO** リストです (図 12-2)。**FIFO** リストは限られたサイズ、又は、無限のサイズを持つ様に定義する事ができます。**FIFO** リストがフルの時は、**producer** はデータを書き込む事はできません。**FIFO** リストが空の場合、**consumer** は待たなければなりません。



図 12-2 メールボックスの使用概念図

セマフォと同様に、メールボックスも **SystemVerilog** のクラスとして実現されています。ハンドルの **new** コンストラクタを使用してオブジェクトを割り当てる手順を取ります。例えば、

```
mailbox fifo;
fifo = new(3);
```

の様に使用します。メールボックスはクラスであるので、表 12-2 の様なメソッドを備えています。

表 12-2 メールボックスのメソッド

メソッド	意味
function new(int bound = 0);	メールボックスを割り当てます。bound が 0 であるとメールボックスは無限のサイズを持ち、producer は決してブロックしません。bound が 0 でない場合、メールボックスは有限のサイズとなります。
function int num();	メールボックスに存在するメッセージ数を戻します。戻す値は、次の get、又は、put まで有効な一時的な値です。
task put(singular message);	メッセージをメールボックス (FIFO) に書き込みます。メールボックスがフルの場合、書き込むプロセスは、メールボックスに空きができるまで待ち状態に入ります。
function int try_put(singular message);	メッセージをメールボックス (FIFO) に書き込みます。但し、ブロックしません。メールボックスに空きがある場合、メッセージを書き込み、正の整数を戻します。メールボックスがフルの場合、0 を戻します。
task get(ref singular message);	メールボックスから 1 メッセージを取り出します。メールボックスが空の場合、取り出すプロセスは、メッセージが届くまで待ち状態になります。
function int try_get(ref singular message);	メールボックスから 1 メッセージを取り出します。但し、ブロックしません。メールボックスが空の場合、0 を戻します。メッセージを正しく取得できれば正の整数

	を戻します。正しく取り出せない場合は、負の整数を戻します。
<code>task peek(ref singular message);</code>	メールボックスから 1 メッセージをコピーします。 メールボックスが空の場合、コピーするプロセスは、メッセージが届くまで待ち状態になります。
<code>function int try_peek(ref singular message);</code>	メールボックスから 1 メッセージをコピーします。但し、ブロックしません。 メールボックスが空の場合、0 を戻します。 メッセージを正しくコピーできれば正の整数を戻します。正しくコピーできない場合は、負の整数を戻します。

`producer` が書き込んだメッセージの型と、`consumer` が取り出すメッセージの型が一致しない場合、取り出しが正しく行えない可能性があります。その様な状況では、実行時に異常終了する事があります。

例 12-3 メールボックスの使用例

サイズが 3 のメールボックスの例を以下に示します。3 個までのメッセージを書くまでは、メールボックスには余裕があり、`put()`メソッドは直ぐに戻りますが、それ以上になると `put()`メソッドは待ち状態に入ります。

```

module test;
    mailbox mb;           // mailbox
    shortint val;

    initial begin
        mb = new(3);
        #10 mb.put (100); $display("@%0t: put val=%0d", $time, 100);
        #10 mb.put (200); $display("@%0t: put val=%0d", $time, 200);
        #10 mb.put (300); $display("@%0t: put val=%0d", $time, 300);
        #10 mb.put (400); $display("@%0t: put val=%0d", $time, 400);
    end
    end
    initial begin
        #50;
        while( mb.num > 0 ) begin
            mb.get (val);
            $display("@%0t: got val=%0d", $time, val);
        end
    end
endmodule

```

実行結果は以下の様になります。

```

@10: put val=100
@20: put val=200
@30: put val=300
@50: got val=100
@50: got val=200
@50: got val=300
@50: put val=400
@50: got val=400

```

値 300 を書き込むとメールボックスがフルになるため、値 400 の書き込みはブロックされません。

```
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

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ブティリスト

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ブティリストは、@(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 の組が変化する場合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、センシティブティリストは@(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;

  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、センシティブティリストが、@(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、センシティブ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的手段

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するタイミングとしては、遅すぎる。
\$monitor	Postponed	無	

例えば、下記の記述は望ましいテストベンチです。下記の\$display 文は、Inactive 領域で実行するので、このテストベンチは bad_dut を正しくない RTL 組み合わせ回路記述であると判断する事ができます。

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

    dut DUT(.*);

    initial begin
        for( int i = 0; i < 4; i++ )
            #10 {a,b} = i;
    end
    initial forever @(a,b)
        #0 $display("@%0t: a=%b b=%b out=%b", $time, a, b, out);
endmodule
```

```

module encoder(input logic [7:0] data,output logic [2:0] code);

    always @(data)
        if( data == 8'b0000_0001 )      code = 0;
        else if( data == 8'b0000_0010 )  code = 1;
        else if( data == 8'b0000_0100 )  code = 2;
        else if( data == 8'b0000_1000 )  code = 3;
        else if( data == 8'b0001_0000 )  code = 4;
        else if( data == 8'b0010_0000 )  code = 5;
        else if( data == 8'b0100_0000 )  code = 6;
        else if( data == 8'b1000_0000 )  code = 7;
        else code = 'x;

endmodule

```

エンコーダーのインスタンスを以下の様に作り、エンコーダーへの入力 `data` を順に生成してテストします。

```

module test;
    logic [2:0] code;
    logic [7:0] data;

    encoder DUT(.*) ;

    initial begin
        $display("      data      code");
        for(int i = 0, val = 1; i < 8; val <= 1, i++ ) begin
            #10 data = val;
        end
    end

    initial forever @(data)
        #0 $display("@%3t: %b      %0d", $time,data,code);
endmodule

```

実行結果は以下の様になり、このテストに関する限りエンコーダーは正しく動作しています。

```

      data      code
@ 10: 00000001    0
@ 20: 00000010    1
@ 30: 00000100    2
@ 40: 00001000    3
@ 50: 00010000    4
@ 60: 00100000    5
@ 70: 01000000    6
@ 80: 10000000    7

```



17.7.2.7 Gray コード

Gray コードをバイナリーコードに変換 (図 17-6) する記述を紹介します。

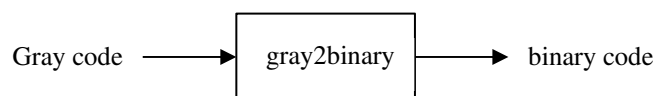


図 17-6 Gray コードをバイナリーコードに変換する回路のブロックダイアグラム

Gray コードは、バイナリーコードと同じビット数を必要としますが、連続するコード間では、

1 ビットしか異ならないという特徴があります。表 17-5 は 4 ビットのバイナリーコードと Gray コードを示しています。Gray コードの適用例としては、カルノー図が挙げられます。

表 17-5 バイナリーコードと Gray コード

binary	Gray	binary	Gray	binary	Gray	binary	Gray
0000	0000	0100	0110	1000	1100	1100	1010
0001	0001	0101	0111	1001	1101	1101	1011
0010	0011	0110	0101	1010	1111	1110	1001
0011	0010	0111	0100	1011	1110	1111	1000

例 17-7 Gray コードをバイナリーコードに変換する記述例

Gray コードを構成するビットを MSB から順に、g3、g2、g1、g0 とし、バイナリーコードを MSB から順に b3、b2、b1、b0 とします。まず、b3 に貢献する Gray コードを集めると表 17-6 の様になります。

表 17-6 b3 を求めるための Gray コード

binary	Gray
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000

以下の様な関係が得られます。以下の式で、a'とは、~a を意味するものとします。また、a&b を ab と書く事にします。そうすると、Gray コードの 1100 は、g3g2g1'g0' と書ける事に注意して下さい。他の Gray コードも同様にして積 (&) として表現する事ができます。

$$b3 = g3g2g1' \mid g3g2g1 \mid g3g2'g1 \mid g3g2'g1' = g3g2 \mid g3g2' = g3$$

同様にして、残りの関係式を以下の様に求めます。

$$\begin{aligned} b2 &= g3'g2g1 \mid g3'g2g1' \mid g3g2'g1 \mid g3g2'g1' = g3'g2 \mid g3g2' = g3 \wedge g2 \\ b1 &= g3'g2'g1 \mid g3'g2g1' \mid g3g2g1 \mid g3g2'g1' \\ &= g3' (g2 \wedge g1) \mid g3 (g2 \wedge g1)' = g3 \wedge (g2 \wedge g1) \\ b0 &= g3'g2'g1'g0 \mid g3'g2'g1g0' \mid g3'g2g1g0 \mid g3'g2g1'g0' \mid \\ &g3g2g1'g0 \mid g3g2g1g0' \mid g3g2'g1g0 \mid g3g2'g1'g0' \\ &= g3'g2' (g1 \wedge g0) \mid g3'g2 (g1 \wedge g0)' \mid g3g2 (g1 \wedge g0) \mid \\ &g3g2' (g1 \wedge g0)' = (g3 \wedge g2)' (g1 \wedge g0) \mid (g3 \wedge g2) (g1 \wedge g0)' \\ &= (g3 \wedge g2) \wedge (g1 \wedge g0) \end{aligned}$$

これらの関係を利用して Gray コードをバイナリーコードに変換します。連続代入文で記述するので、出力ポートは wire で十分です。従って、output に対して logic の指定は不要です。

```
module gray2binary(input g3,g2,g1,g0,output b3,b2,b1,b0);
assign b3 = g3;
assign b2 = g3^g2;
assign b1 = g3^(g2^g1);
assign b0 = (g3^g2)^(g1^g0);
endmodule
```



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 組み合わせ回路により、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。 ④ 出力は、クロックと状態の遷移に同期します。
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。

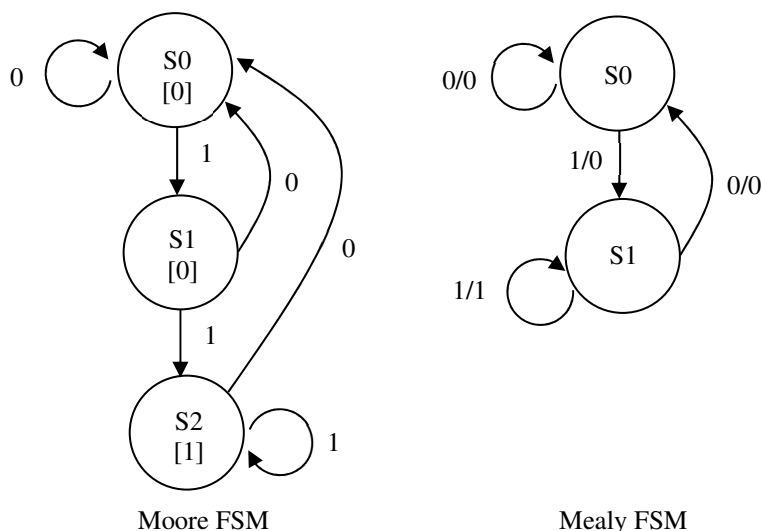


図 17-15 連続した 2 つの 1 を認識する FSM の状態遷移図

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、1 の数が偶数であれば、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。

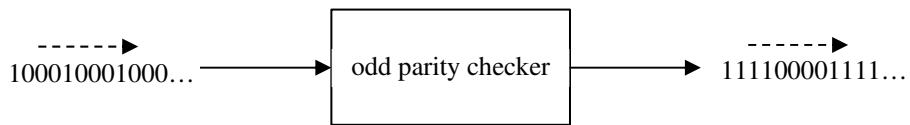


図 17-16 パリティチェッカーのブロックダイアグラム

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。

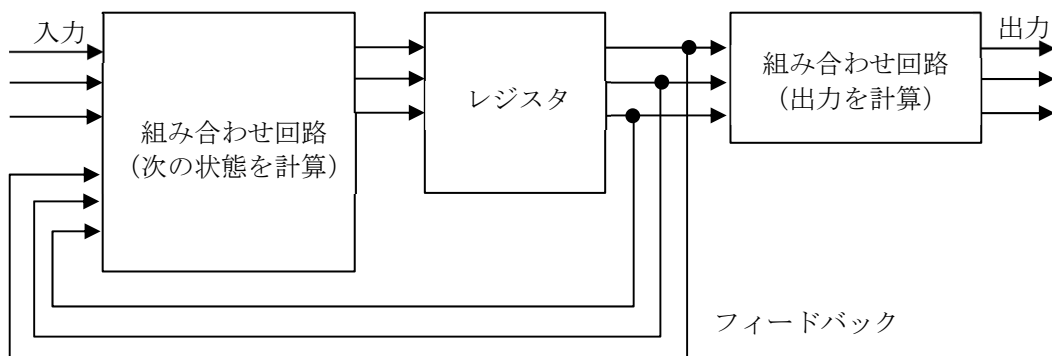


図 17-17 Moore FSM ([9])

Moore FSM のモデリングは、以下の様な構造を取ります。

```

module moore_fsm(input clk, reset, ..., output logic out);
state_e state;
    always @(posedge clk, posedge reset)
    if( reset )
        state <= S0;
    else
        case (state)
        S0: state <= ...
        S1: state <= ...
        ...
        endcase

    always @(state)
    case (state)
    S0: out = ...;
    S1: out = ...;
    endcase
endmodule
    
```

現在の状態を示す変数を宣言する。

次の状態を計算して、レジスタに保存する。ここでは、出力の計算を行わない。

出力を計算する。

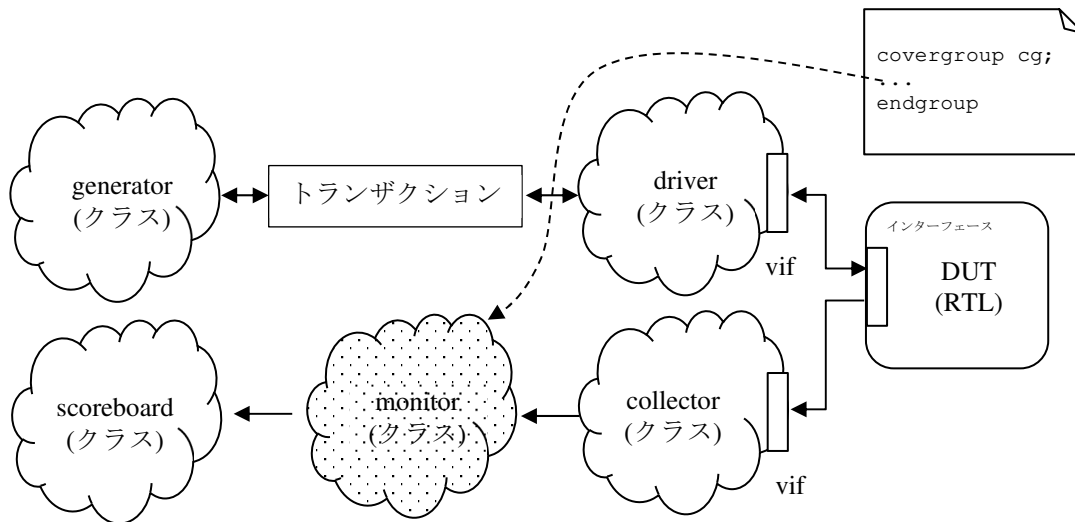


図 21-2 モニターに於けるカバレッジ計算

近年の検証手法では、ドライバーは DUT をドライブする目的に使用され、他の検証コンポーネント、例えば、コレクター、モニター、スコアボード、チェッカー等が DUT からのレスポンスをチェックする役目を持ちます。これらの検証コンポーネントが動作する時には、トランザクションには検証に使用された全ての情報が記録されています。従って、この時点のトランザクションはカバレッジ計算に適しています。ドライバーが、トランザクションを取得した時点では、DUT への入力しか決定されていないため、カバレッジ収集には不完全な状態です。

21.5.2 カバーグループの定義

カバレッジ計算をするためには、カバーグループを定義しなければなりません。カバレッジ計算は、トランザクションに定義されている項目に関するカバレッジを計算する目的を持つので、本来は、トランザクション内にカバーグループを定義するのが妥当であると考えられます。然し、トランザクションは一時的なオブジェクトであり、DUT をドライブした後は消滅してしまいます。従って、トランザクション内にカバーグループを定義するのは適切な方法とは言えません。

一方、コレクター、モニター、スコアボード、チェッカー等の検証コンポーネントは、シミュレーション中は常時存在するので、カバーグループを定義するためのコンテナとしての資格を備えています。コレクターは、DUT からのレスポンスからトランザクションに逆変換する機能だけに専念する方が明確であると考えられます。モニターは、トランザクションを検証コンポーネントに伝達する役目を持ちますが、簡単なチェックとカバレッジ計算も行えます。一方、スコアボードは DUT からのレスポンスを詳細に解析する役目を持ち、カバレッジ計算もその一つとなります。チェッカーも同様に詳細なチェックを行います。

以下では、モニター内にカバーグループを定義してカバレッジ計算をする方針を採用します。勿論、コレクターやスコアボードにカバグループを定義する事も可能です。

21.5.3 カバレッジのサンプリング

既に述べた様に、カバレッジの対象項目はジェネレータが生成したトランザクションに定義されています。一方、カバーグループはトランザクション内に定義されていないため、カバーポイントをモニター外から取得しなければなりません。それ故、モニター内のカバーグループにサンプリング関数 `sample()` を定義するとカバレッジ計算は柔軟性が増します。

21.5.4 自動カバレッジ収集例

以上述べた方針を図 21-3 の様な検証環境で実践してみます。

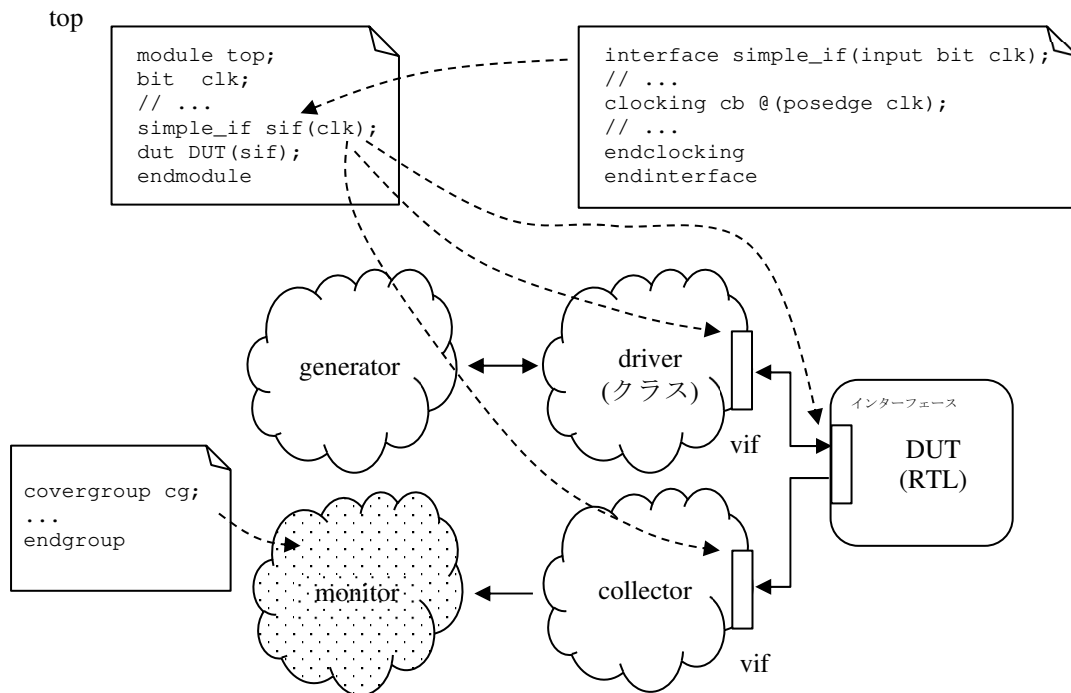


図 21-3 自動カバレッジ収集環境

この検証で使用するクラス、及び DUT を表 21-4 にまとめます。

表 21-4 検証環境を構成する要素

クラス及び DUT	説明
simple_if	DUT をドライブするために必要なインターフェースです。
simple_item	トランザクションを定義するためのクラスです。
simple_component	検証コンポーネントのベースクラスです。
simple_driver	ドライバーのクラスです。
simple_generator	ジェネレータのクラスです。
simple_collector	コレクターのクラスです。
simple_monitor	モニターのクラスです。
dut	簡単な加算器です。
top	トップモジュールです。

21.5.4.1 simple_if

インターフェースには、DUT のポートに対応する変数とクロックを定義します。クロッキングブロックも定義しておきます。

```

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

clocking cb @(posedge clk); endclocking

endinterface

```

21.5.4.2 simple_item

トランザクションには、DUT のポートに対応する変数を定義しておきます。そして、DUT の