Artgraphics

SystemVerilog 超入門

~始めて学ぶ設計のためのハードウェア記述言語 ~

Document Identification Number: ARTG-TD-006-2020
Document Revision: 1.9, 2022.05.28
アートグラフィックス
篠塚一也

SystemVerilog 超入門
~ 始めて学ぶ設計のためのハードウェア記述言語 ~

〒124-0012 東京都葛飾区立石 8-14-1
www.artgraphics.co.jp

Beginner’s Guide to SystemVerilog

8-14-1, Tateishi, Katsushika-ku, Tokyo, 124-0012 Japan
www.artgraphics.co.jp

はじめに

本書は、設計分野で必要とされる SystemVerilog の基礎知識を重点的に解説した初心者のための資料です。また、SystemVerilog とはどのような機能を持つ言語であるかを短期間に、しかも正確に知りたい人のために書かれた資料でもあります。特に、予備知識が全くない技術者や管理者の方々にもお薦めします。しかし、表題にある「超入門」は「超易しく書かれた入門書」を意味するわけではありません。寧ろ、「超詳しく書かれた入門書」の意味です。本書を通して、読者は SystemVerilog の全体像を把握しつつ、SystemVerilog の設計機能に集中して基礎知識の理解を深める事ができます。特に、SystemVerilog による記述がどの様なハードウェアに変換されるかを正しく理解するための素養を身に付ける事ができます。

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）として公開されましたが、LRM は 1300 ページにもおよぶ大作です。そればかりではなく、LRM は難解な英文でしたためられているため、誰でもが誤解なく理解する事ができる代物ではありません。特に、初心者にとって LRM は最初の学習書としての役割には不適切であるといえます。本書は、LRM の精神を尊重しつつ、初心者に必要な SystemVerilog の基礎知識を詳しく解説した資料です。初心者にとっての分かり易さを本書の最重要課題としているので、おのずから、本書がカバーする言語仕様書としての範囲も限定されています。例えば、プロセス間通信機能、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、アサーション等の機能解説は割愛せざるを得ませんでした。これらの機能は、主として、検証分野で利用されている現状を考えれば、合理的な判断であるとの賛同を得られると思います。

しかし、検証に関わるすべての機能が割愛されているわけではありません。例えば、主として検証分野で使用されるデータタイプに関しての詳しい解説も含まれています。あるいは、SystemVerilog で記述したデザインを検証するために必要な機能、技術、手法等の解説も含まれています。要約すると、一般的に検証機能と呼ばれるカテゴリーに属する機能の解説を省き、SystemVerilog への初心者が把握すべき基礎知識を重点的に解説しています。したがって、本書を読了後は、初心者はもはや初心者ではなく、SystemVerilog の設計機能を縦横無尽に使いこなせる技術者になっています。そして、検証を専門に志す技術者は、LRM または巻末の他の文献で更に上級者向けの知識習得へと進む事ができます。

本書は、単に SystemVerilog の設計機能を解説しているだけではありません。SystemVerilog の備えている機能をロジック設計に的確に応用するための知識を提供する目的も持っています。そのためには、SystemVerilog で記述されたデザインがどの様なハードウェアに変換されるかを具体例により解説しています。その際、RTL 論理合成ツールの使用は暗黙の了解となっています。しかし、本書では特別な論路合成ツールを仮定せずに、一般的な論理合成規約にしたがい解説を進めています。初めてハードウェア記述言語を学ぶ人にとって、最も難解な過程は、どのような記述をすればマルチプレクサ、フリップフロップ、ラッチ、デコーダ、エンコーダ等の回路を実現できるかを理解する事であると思います。本書の随所で、基本的な回路と SystemVerilog 記述との対応を示しているのは、その学習過程を支援する役割の一つです。

本書は、概要を含めて 17 章から構成されています。第 1 章の概要では、SystemVerilog の目的とハードウェアモデリングの概要を解説しています。総じて、この章の難易度は高いのですが、基本的な回路と SystemVerilog 記述との対応を真理値表により解説しているので、設計知識を持つ技術者の興味を誘う話題で満ちています。この章の内容を理解すると SystemVerilog の使用法に興味がわくことは確実と言えます。

第 2 章から第 14 章までは SystemVerilog の基礎知識の解説に割かれています。データタイプの解説から始まり、各種アレイの解説、オペレータの種類、実行文の記述法、タスク、ファンクション、インターフェース、パッケージ、モジュール、クラス等に関する基礎知識の解説が含まれています。これらの章の内容は丹念に読まれる事が期待されています。解説だけでなく多くの記述例が示されているので、手を動かしながら学習を進めて下さい。

第 15 章のシミュレーション実行モデルは SystemVerilog の最も重要な規約であるので、できるだけ早い時期に目を通すようにして下さい。一読では内容を理解する事は難しいと思えますが、一読もしないと本書全体の内容を正しく理解する事は困難になります。デザインをシミュレーションするためには、シミュレーションの論理に関する正しい理解が必要です。

第 16 章は、and、or、not 等のゲートプリミティブに関する基本的な解説を含んでいます。SystemVerilog で高位の記述をする場合には、これらのゲートは必要のない機能ですが、組み合わせ回路等をデザインする際、時として必要になります。例えば、グルーロジックを例としてあげられます。

第 17 章は、SystemVerilog に関する言語規約等をまとめた補足的な内容を含んでいます。また、この章には符号付き整数と 2 の補数表現についての解説も含まれています。補足説明により、SystemVerilog の signed と unsigned の相違を明確に理解する事ができます。

本書の記述の中には、初心者にはやや難易度が高いと思われる内容も含まれています。そのような内容を含む章や節のタイトルには＊印をつけて明記してあります。それらの内容に遭遇した際には、初読時では後回しにするか概略を読み取る程度にして、本書全体を理解した後に再学習して下さい。

アートグラフィックス
篠塚一也

変更履歴

日付	Revision	変更点
2020.11.30	1.0	初版。
2021.01.15	1.1	使用例の解説を追加。
2021.02.22	1.2	ゲートプリミティブの記述を追加。
2021.03.12	1.3	全体的な校正を実施。
2021.04.30	1.4	プログラムとチェッカーの記述を追加。
2021.09.30	1.5	全体的に説明を補足。
2021.12.15	1.6	演算に関する説明補足。
2022.02.08	1.7	① 1.8 節にデザインの記述法を追加。 ② 4.2 節に演算精度の関する記述を追加。
2022.03.08	1.8	第 1 章を改訂。
2022.05.28	1.9	全体的に校正を実施。

目次

本書で使用する略語一覧

略語	定義
ALU	Arithmetic Logic Unit
BNF	Backus-Naur Form
DPI	Direct Programming Interface
DUT	Design Under Test、または、Device Under Test
EDA	Electronic Design Automation
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
LHS	Left Hand Side
LRM	Language Reference Manual、すなわち、IEEE Std 1800-2017
LSB	Least Significant Bit
MOS	Metal-Oxide-Silicon、または Metal-Oxide-Semiconductor
MSB	Most Significant Bit
RTL	Register Transfer Level
UDP	User-Defined Primitive
UVM	Universal Verification Methodology

1 概要

1.1 SystemVerilog の歴史

SystemVerilog は、Verilog HDL を拡張した言語ですが、実際には、その他の多く言語から影響を受けています。設計分野においては、SUPERLOG、および C、検証分野においては、SUPERLOG、VERA C、C++、VHDL、OVA、PSL 等の影響を受けています。

SystemVerilog 言語仕様の主要部分は SUPERLOG を基にして Accellera standard groups により開発され、2002 年に SystemVerilog 3.0 としてリリースされました。SystemVerilog 3.0 は第三世代の Verilog 言語として位置付けられています。第一世代は、良く知られている Verilog-1995 で、第二世代は Verilog-2001 です（表 1-1）。

表 1-1　世代別言語仕様

言語の世代	言語
第一世代	Verilog-1995
第二世代	Verilog-2001
第三世代	SystemVerilog 3.0

SystemVerilog 3.0 は、その後、Accellera による SystemVerilog 3.1、および SystemVerilog 3.1a 等の改訂版を経て、2005 年 11 月に IEEE Std 1800-2005 として正式に公開されました。これが初版の SystemVerilog 言語仕様です。SystemVerilog 言語仕様は数年に一度の改訂が行われており、年代的な言語の変遷を図 1-1 にまとめました。本書の解説は、IEEE Std 1800-2017 に準拠しています。

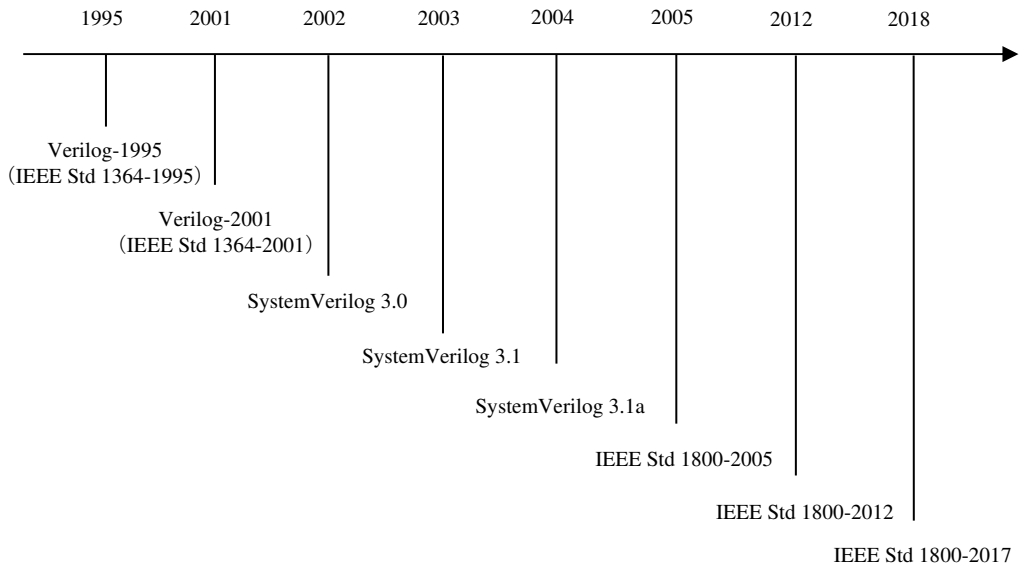

図 1-1　SystemVerilog 言語仕様の変遷

1.2 SystemVerilog の概要

SystemVerilog はハードウェアを記述するための言語であるだけでなく、記述した内容を検証する機能も備えています。ハードウェアを記述した内容は、一般的には、デザインと呼ばれます。デザインを検証する際、ハードウェア仕様を基に動作を確認するため、正確で、かつ誤解がないようにハードウェア仕様を表現しなければなりません。SystemVerilog にはハードウェア仕様を記述する機能も備えられているので、SystemVerilog を使用する事によりハードウェア設計、仕様記述、検証作業を統一的に表現する事ができます（図 1-2）。したがって、

デザインと検証間でのハードウェア情報授受は円滑に進められます。SystemVerilog は Verilog や VHDL と共に HDL と呼ばれる言語のカテゴリーに属します。本書は SystemVerilog を対象にして解説を進めますが、多くの概念は HDL 全体に共通します。それらの共通的な概念を強調する際には、本書では SystemVerilog という代わりに HDL と表現する事があります。

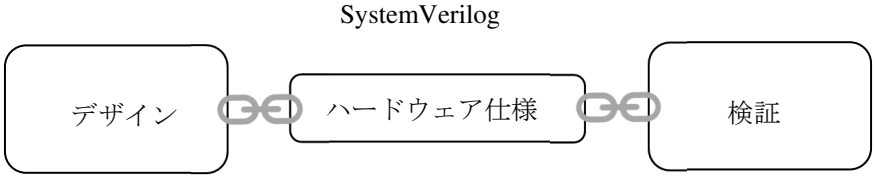

図 1-2　SystemVerilog の機能範囲

検証に必要とされる主な機能としては、プロセス間通信機能、クラス、プログラム、インターフェース、ランダムスティミュラス生成、ファンクショナルカバレッジ、アサーション等がありますが、本書は SystemVerilog による設計をするための基礎知識を提供する事を目的にしているため、これらの検証機能の解説は一部を除いて割愛します。

1.3　SystemVerilog によるハードウェアモデリング

自然言語による表現と異なり、SystemVerilog による表現は曖昧性の無い記述を導きます。記述された実装内容は、様々な変換過程を経て、ゲートレベルで実現されます。ゲートレベルの表現は、一般的にはネットリストと呼ばれ、テクノロジーライブラリーのセルをネット（配線を示すワイア）で接続した膨大な量の記述表現となります。このゲートレベルは、セルインスタンスをネットで結合しているので、HDL の表現形式の一つと言えますが、実質的には、異なる言語による物理表現とも言えます。本書の目的は、ゲートレベルよりも高位のレベル、具体的には、RTL、およびそれよりも高位のレベルでデザインを表現する事です。

SystemVerilog で記述された RTL デザインは、シミュレーションを通して機能的に正しい事が確認されます。その後、論理合成や自動配置配線等の EDA ツール[1]を使用して物理設計が行われ、最終的には、ウエハ上に集積回路として実現されます。本書では、論理合成の過程以前の機能的な設計過程が主題になります。特に、SystemVerilog を使用してハードウェア仕様を実現するために必要な基礎知識を解説します。

1.4　論理合成とシミュレーション

シミュレーションは、一般的には、機能的な検証やタイミングの検証を行う過程です。しかし、RTL ではクロック信号に同期してデザインが機能的に正しく動作する事を確認します。シミュレーションはイベントドリブン方式を採用し、イベントが発生する事により論理のシミュレーションが進行していきます。ここで、イベントとは、信号値が変化する状態を意味します。SystemVerilog では、信号が単純に変化する状態と特別な値に変化する状態を区別して表現します。例えば、イベントが発生する状態を `@(expr)` で表現すると、式 `expr` で示されている何れかの信号の指定した状態が発生するまで後続の命令をブロックします。つまり、指定したイベントが発生すると記述した動作が実行するという機能を表現できます。

以下に示す記述は信号 `a` と `b` を入力に持つ二入力 `AND` 回路を表現しています。記述によれば `a` と `b` の何れかが変化すると、オペレーション `out = a&b` が実行されます。そして、この動作は常時（`always`）繰り返します。この記述は `AND` ゲートに論理合成されますが、動作記述をシミュレーションした結果と論理合成された表現のシミュレーション結果は一致しなければなりません。

[1] 本書では EDA ツールと表現する代わりにソフトウェアツールとも呼びます。

- a==0 であると、out==b である。
- a==1 であると、out==1 である。

したがって、2 入力の OR 回路を以下のような動作記述で表現する事ができます。

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

always @(a,b)
    if( a == 1'b1 )
        out = 1;
    else
        out = b;

endmodule
```

以上の解説から、ハードウェア仕様を満たす実装表現法は一意的に定まらない事が分かります。この章では仕様を基にして真理値表からハードウェア動作記述を導きましたが、実際には様々な方法により最適な表現を実現します。その際、表現する方法としてどのような選択肢があるかの知識が的確な動作記述を導きます。その知識を提供するのが本書の役割です。

1.7 階層設計と非階層設計

複雑なデザインは階層設計で行われます。特に、組み合わせ回路においては小規模な回路でも階層的に設計する事がしばしばあります。この節では、デザインを階層的に記述する方法と階層設計を用いない記述による方法を比較します。例としては、簡単な 4 ビットのリップルキャリーアダーを使用する事にします。

例 1-7　階層設計による 4 ビットのリップルキャリーアダー記述例

先ず、階層設計のデザインにおけるビルディングブロックとしての役割を果たすフルアダーを以下のように準備します。

```
module fa(input a,b,ci,output co,sum);
assign {co,sum} = a+b+ci;
endmodule
```

フルアダーを組み合わせて 4 ビットのリップルキャリーアダー（rca）を図 1-10 のように構築します。

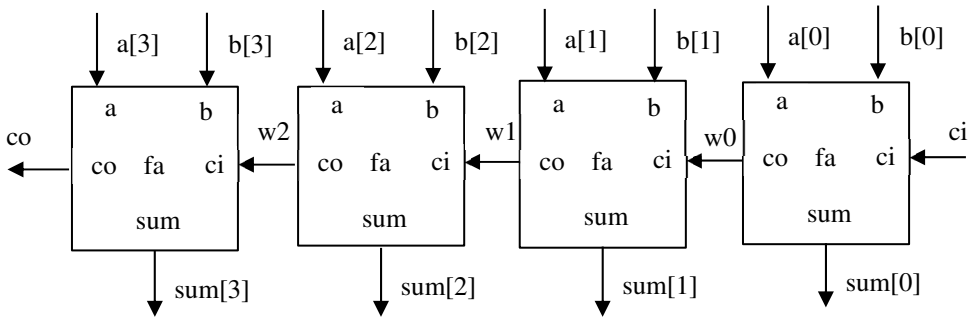

図 1-10　rca の構成

このように構築されたリップルキャリーアダーに対して、SystemVerilog による記述は以下のようになります。3 つのネット w0、w1、w2 がキャリーとして使用されています。

```
module rca(input [3:0] a,b,logic ci,output co,[3:0] sum);
wire   w0, w1, w2;
```

```
fa FA0(a[0],b[0],ci,w0,sum[0]);
fa FA1(a[1],b[1],w0,w1,sum[1]);
fa FA2(a[2],b[2],w1,w2,sum[2]);
fa FA3(a[3],b[3],w2,co,sum[3]);

endmodule
```

■

さて、このように階層的に構築する方法以外にも rca と同等な機能を持つ組み合わせ回路を実現する事ができます。例えば、以下のように 4 ビットのフルアダーを実装する事ができます。

例 1-8　階層を用いない 4 ビットのフルアダー記述例

以下に示す方式では、フルアダーfa を使用していないので階層設計にはなっていません。にもかかわらず、rca と fa4 は全く同様な機能を持つ事に注意して下さい。

```
module fa4(input [3:0] a,b,logic ci,output co,[3:0] sum);
assign {co,sum} = a+b+ci;
endmodule
```

しかし、この記述方式にはハードコーディングがあり柔軟性に欠けています。一般的には、下記のようにビット数をパラメータ化して汎用性を高めます。

```
module fa4 #(NBITS=4)
    (input [NBITS-1:0] a,b,logic ci,output co,[NBITS-1:0] sum);
assign {co,sum} = a+b+ci;
endmodule
```

■

両方式の差異を観察すると、非階層設計では 1 ビットと 4 ビットの違いを SystemVerilog 記述の中に吸収できる点を活用しています。例えば、ここで紹介したフルアダーfa と fa4 との違いは入出力のビット数だけで動作記述は同じです。一方、階層設計ではビット数を厳密に意識する必要があります。両者の構築法の差異を表 1-11 にまとめておきます。

表 1-11　rca に関する階層設計と非階層設計の比較

比較項目	階層設計	非階層設計
ビルディングブロック	フルアダーを定義する必要がある。	必要がない。
回路の構築	ネットを使用してビルディングブロックを接続する。しかも、ビット数だけフルアダーのインスタンスを作らなければならない。	アルゴリズムを式で表現するだけで良いので、ネットを意識する必要はない。つまり、実装方式等を意識する必要はない。

参考 1-1

ここで紹介した例によれば、階層設計で構築した rca には利点がないように思えますが、実はその判断は正しくない事が分かります。例えば、fa4 の記述ではキャリーの操作ができませんが、階層設計の rca ではそれぞれのビットに関するキャリーを操作する事ができます。この利点により計算のオーバーフローを厳密に検知する事ができるようになります。詳細な解説は、17.4.4 項を参照して下さい。

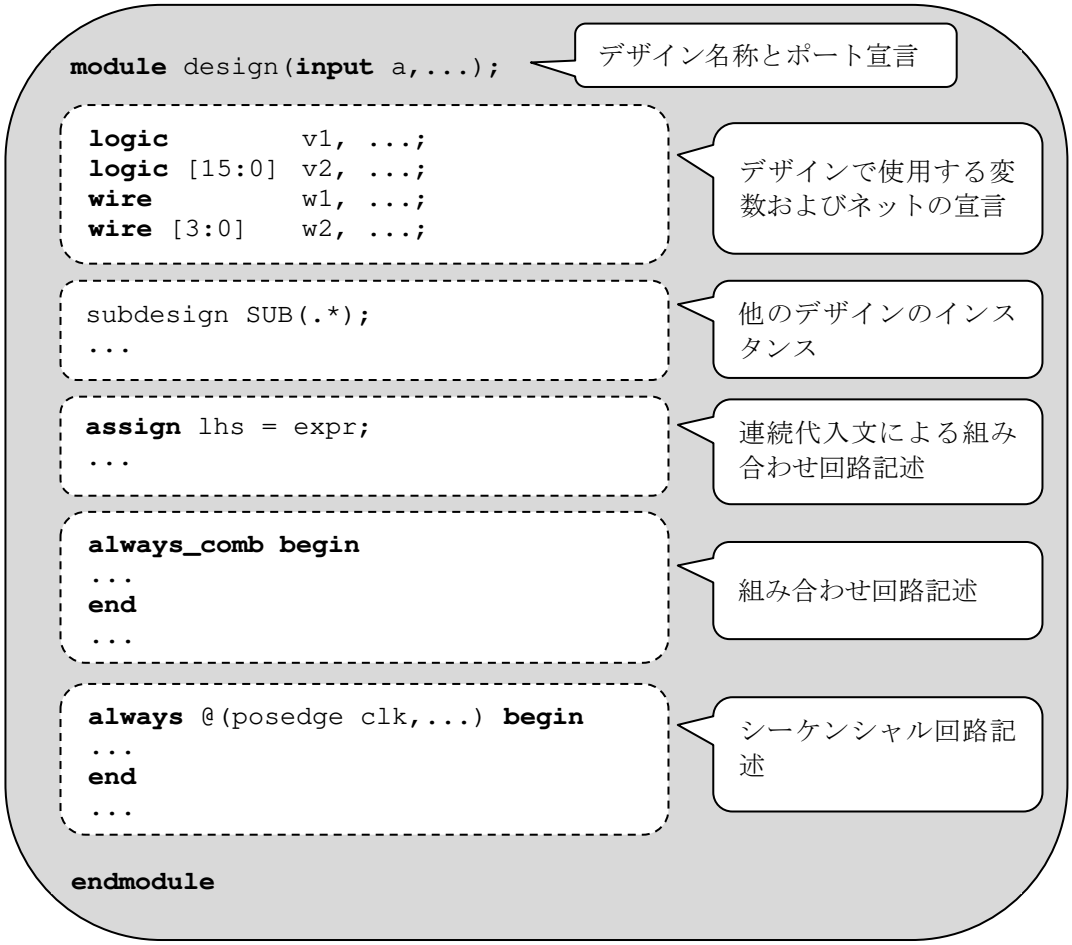

図 1-12　SystemVerilog によるデザインの構造

デザインの記述は、キーワード module で始まり、もう一つのキーワード endmodule で終了します。二つのキーワード間にデザインの実装情報を定義しなければなりません。実装情報としては、以下のような宣言と定義が必要です。

- デザイン名称とポート宣言
- デザインで使用する変数とネットの宣言
- 組み合わせ回路を記述する連続代入文
- 組み合わせ回路記述
- シーケンシャル回路記述
- 他のデザインのインスタンス
- 記述で使用するタスクやファンクションの定義

これらの宣言と定義を完成するためには、第 2 章以降で解説するデータタイプ、オペレータ、式の記述法、モジュールの定義法等の知識が必要になります。

1.9　本書でのシンタックス記述法

LRM では、標準的な BNF を使用してシンタックスを記述しています。なお、一般的な BNF の定義は言語関係の書物（例えば、[4]）を参照して下さい。本書では、そのままの形で LRM からシンタックスを借用します。例えば、モジュールのシンタックスは以下のように表現されます（[1]）。

```
module_declaration ::=
    module_nonansi_header [ timeunits_declaration ]
```

2 データタイプ

データタイプとは、値の集合とそれらの値に適用されるオペレーションの集合です。例えば、以下の記述には logic データタイプが使用され、logic に定義されている標準的なオペレーションを使用して動作が表現されています。

```
logic [3:0]   a, b, sum;
logic         co;
...
always @(a,b)
    {co,sum} = a+b;
```

logic [3:0]は 4 ビット符号なし整数を表現します

logic には標準的なオペレーションが定義されている

SystemVerilog には、Verilog にはない次のようなデータタイプが備えられています。C/C++に見られるデータタイプの他にプロセス間通信で使用されるデータタイプが追加されています。

- logic
- bit
- byte
- shortint
- int
- longint
- shortreal
- string
- enum
- struct
- union
- class
- semaphore
- mailbox
- process

ちなみに、Verilog のデータタイプには以下の種類があり SystemVerilog でも使用できます。ただし、event は SystemVerilog において機能が大きく拡張されています。

- reg
- integer
- time
- real
- realtime
- event

本書では主として設計時に使用するデータタイプを解説しますが、デザインの検証時に良く使われる便利なデータタイプの解説も含みます。ただし、クラスは非常に重要なデータタイプであり多くの機能を含むので、他のデータタイプとは別に第 12 章で詳しく解説します。なお、データタイプに関する更に詳細な解説は、巻末の文献を参考にして下さい。

2.1 データタイプとデータオブジェクト

SystemVerilog は、オブジェクトとデータタイプを明確に分離します。表 2-1 はデータタイプの使用例を示しています。データタイプは、データオブジェクトを宣言する目的のため、または既存のデータタイプから新たなデータタイプを定義するために使用されます。

表 2-1　データタイプの使用例

宣言	値の集合	オペレーション
`logic [31:0] a;`	変数 a は 32 ビットの符号なし整数。	四則演算に加えてビット演算等を適用する事ができる。
`byte b;`	変数 b は 8 ビットの符号付き整数。	
`real r;`	変数 r は 64 ビットの実数。	四則演算が定義されている。

データオブジェクトは、値とデータタイプが付与された名称付きのインスタンスです。パラメータ、変数、およびネットがデータオブジェクトの良い例です。

例 2-1　データオブジェクトの例

以下の記述では、`PI` は実数 3.14 を示す定数のオブジェクトです。そして、`state` は `int` 型の変数を示すオブジェクトです。

```
parameter real PI = 3.14;
int       state;
```

■

2.2　論理値

SystemVerilog では、表 2-2 に示すような 4 つの論理値が定義されています。

表 2-2　SystemVerilog の論理値

論理値	意味
0	値 0 を表現し、偽の意味を持ちます。
1	値 1 を表現し、真の意味を持ちます。
x	unknown という論理値を表現します。
z	high-impedance を表現し、接続を遮断する場合等に使用されます。

これらの論理値を表現するためのデータタイプは、`logic` です。例えば、次の `if` 文における条件は、`s` が真であるかどうかを判定しています。 `s` が 0、`x`、または `z` であると条件は偽になり、`else` クローズが実行します。

```
logic  a, b, s, q;
...
always @(a,b,s)
   if( s )
      q = a;
   else
      q = b;
```

これは、以下の記述と同等な表現です。

```
logic  a, b, s, q;
...
always @(a,b,s)
   if( s == 1'b1 )
      q = a;
   else
      q = b;

endmodule
```

3　メンバーで構成されるデータタイプ

本章では、複数のメンバーから構成される以下のようなデータタイプを解説します。

- ストラクチャ
- ユニオン
- アレイ

ストラクチャとユニオンは、個々のメンバーに固有の名称を付けて操作をする事ができる便利なデータタイプです。ストラクチャのメンバーは独立した位置を保有しますが、ユニオンのメンバーは論理的に同じ位置を共有します。アレイは、各メンバーが同じデータタイプを持ち、メンバーへのアクセスにはインデックスまたはキーを使用します。アレイには、固定サイズを持つタイプと実行時にサイズを決定する事ができるタイプがあります。表 3-1 は、これらのデータタイプのまとめです。

表 3-1　ストラクチャ、ユニオン、アレイの特徴

データタイプ	メンバーの位置	特徴
ストラクチャ	各メンバーは固有の位置を確保する。	• メンバーに名称を付けて管理する事ができる。 • 各メンバーには個別のデータタイプを定義する事ができる。 • 全体を一つのグループとして扱える packed タイプがある。
ユニオン	メンバーは論理的に同じ位置に配置される。ただし、各メンバーの大きさは異なっても良い。	
アレイ	各メンバーは固有の位置を確保する。	• メンバーには名称を与えられないが、インデックスまたはキーでメンバーにアクセスする事ができる。 • メンバーは同じデータタイプを持つ。 • 固定サイズのアレイと実行時にサイズを決定するアレイが存在する。 • アレイの要素を操作するための便利なメソッドが提供されている。

設計では、主として、固定サイズを持つアレイが使用されますが、実行時にアレイのサイズを調節する事ができるダイナミックアレイ、associative アレイ、キュー等はテストベンチを記述する際に便利な手段です。本章では、これらの可変長アレイについても解説します。

3.1　ストラクチャ

ストラクチャは複数のメンバーを一つのグループとしてまとめたデータタイプです。各メンバーは異なるデータタイプを持つ事ができます。メンバーを個別に参照、または操作する事ができますが、グループ全体として参照する事もできます。以下のシンタックス（[1]）を使用してストラクチャを定義します。キーワード `struct` で始まり、メンバーを `{...}` 内に指定します。

```
struct [ packed [ signing ] ] { struct_member { struct_member } }
    { packed_dimension }

signing ::= signed | unsigned
```

シンタックスが示すように、ストラクチャには `packed` ストラクチャと `unpacked` ストラクチャがあります。修飾子 `packed` を明示的に指定しない場合、ストラクチャは `unpacked` に

なります。struct_member で個々のメンバーを定義します。また、packed_dimension でストラクチャ全体のビット構成を定義する事ができます。packed_dimension を指定する際は、packed ストラクチャでなければなりません。packed ストラクチャに対しては、signed で符号を付ける事ができます。

packed ストラクチャでは全てのメンバーが連続したメモリーに配置され、ストラクチャ全体を一つの数値として扱う事ができます。その代り、packed ストラクチャのメンバーは integral データタイプでなければなりません。unpacked ストラクチャの場合には、メンバーが連続したメモリーに配置される保証はありません。各メンバーを個別に参照する事はできますが、初期化処理を除き全体として扱う事はできません。ストラクチャの種別を表 3-2 にまとめておきます。

表 3-2　packed ストラクチャと unpacked ストラクチャの差異

ストラクチャのタイプ	メンバーの配置と特徴	制限
packed	• メンバーは連続した領域に配置される。 • 最初のメンバーが MSB の位置になるように配置される。 • ストラクチャ全体を一つの数値として扱う事ができる。	メンバーは integral データタイプでなければならない。
unpacked	• メンバーが連続したメモリーに配置される保証はない。 • 初期化処理を除き全体として扱う事はできない。	メンバーには個別にデータタイプを定義する事ができる。

参考 3-1

ストラクチャは、一般的に、論理合成可能なデータタイプです。特に、packed ストラクチャは、packed アレイとして扱う事ができる便利な手段です。
□

例えば、ストラクチャを以下のように使用します。

```
struct { bit [7:0] r, g, b; } color_code;
```

ストラクチャのデータタイプを持つ変数 color_code を直接定義しているので、anonymous ストラクチャと呼ばれる

ストラクチャを使用して新しいデータタイプを定義している

```
typedef struct { logic a, b, co, sum; } ha_arg_s;
typedef struct { string proc_name; int elapsed; } run_statics_s;
```

最初の行は、color_code をストラクチャのデータタイプを持つ変数として定義しています。他の行では、ストラクチャを使用して新しいデータタイプを定義しています。このように定義されたデータタイプで変数を宣言する事ができます。次に、これらの宣言を使用した例を紹介します。

例 3-1　ストラクチャの使用例

以下に示すようにストラクチャを使用する事ができます。

5　代入文

本章では、以下の代入文を解説します。

- 連続代入文
- ビヘイビア代入文

これらの文を使用して組み合わせ回路やシーケンシャル回路を記述します。例えば、以下の記述は簡単な組み合わせ回路の例ですが、連続代入文とビヘイビア代入が使用されています。

```
module dut(input a,b,c,d,e,s2,s1,s0,output logic out);
logic  tmp;

assign out = s2 == 1 ? a : tmp;

always @(s1,s0,b,c,d,e)
    case ({s1,s0})
    0:  tmp = b;
    1:  tmp = c;
    2:  tmp = d;
    3:  tmp = e;
    default   tmp = 1'bx;
    endcase

endmodule
```

連続代入文

ビヘイビア代入文

この記述は、図 5-1 のような回路構成を表現しています。

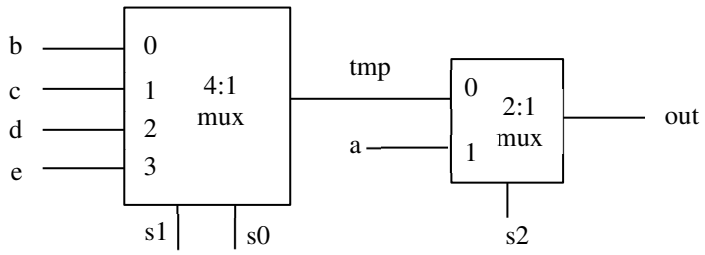

図 5-1　簡単な組み合わせ回路の例

連続代入文はキーワード assign を使用して左辺（ネットまたは変数）と右辺を=オペレータで結びつけた代入文です。連続代入文は右辺の値を左辺に常時割り当てます。すなわち、右辺に使用されている信号の変化はディレーに従って左辺に反映されます。図 5-2 で示すように、右辺は組み合わせ回路に相当し、左辺はその回路の出力に接続されている状態を表現します。

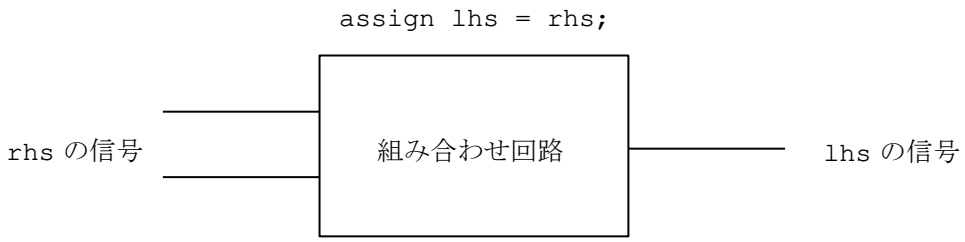

図 5-2　連続代入文の効果

例えば、以下のような連続代入文は、図 5-3 のような組み合わせ回路を生成します。この場

6　プロセス

SystemVerilog はハードウェアを記述するための並列処理言語です。並列処理は、モジュール内に記述する initial プロシージャ、および always プロシージャによって実現されます（図 6-1）。ただし、initial プロシージャは一度しか実行されないので、デザインで使用される事はありません。これらのプロシージャは標準的なプロセスとしてシミュレータにより起動され、ユーザが起動する事はできません。ユーザがプロセスを生成するためには、fork ブロックの機能を利用します。この機能を利用する事により、標準プロセスから任意数の子プロセスを生成する事が可能になります。本書では、プロシージャが実行環境を与えられて実行するとき、プロセスと表現ずることにします。ただし、プロセスの一部の小さな実行単位を特定する場合には、スレッドとも表現する事にします。プロセスは、いつも活動しているとは限りません。通常は、プロセスにイベント待ち、またはディレー制御を含みます。

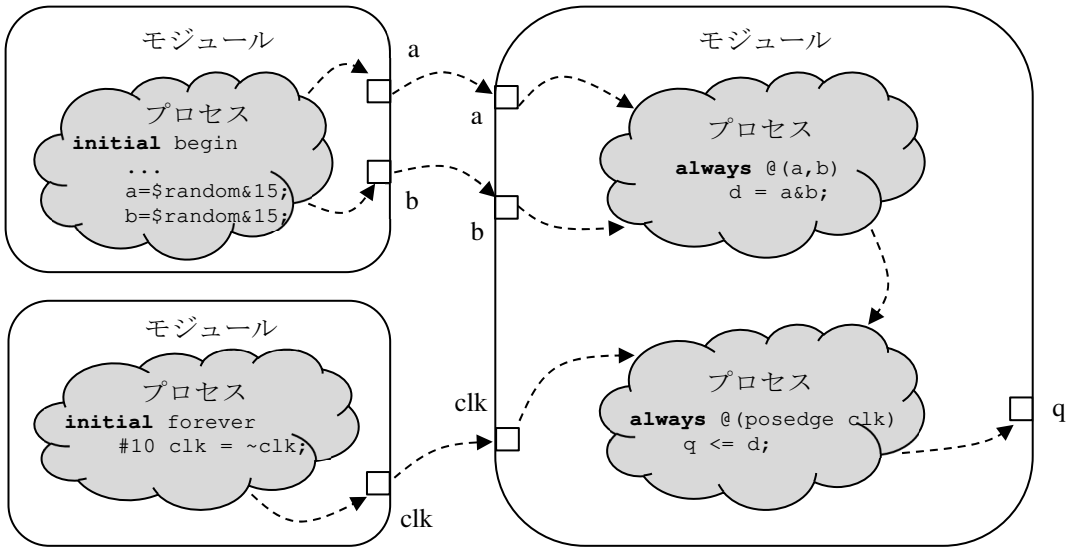

図 6-1　SystemVerilog による標準的なプロセス

図 6-1 は、標準的に生成されたプロセスが並列に実行してハードウェアの動作を表現している様子を示しています。ここで、clk はクロック信号を意味します。破線は信号値の流れを表現していますが、同時に進行している事に注意して下さい。例えば、回路からの出力 q の値は、回路への現在の入力 a と b により決定されていません。クロッキングイベントが発生した時の a と b の値により q の値が決定されています。クロッキングイベント後に起きた a と b の値の変化は現在の q に影響を与えませんが、次の時刻のクロッキングイベントにおける q の値に影響を持ちます。また、q の値をどのようにしてサンプリングするかも難しい課題になります。例えば、クロッキングイベントが起きた時点では、q の値はまだ更新されていません。

ハードウェアを構成する個々のコンポーネントは、元来、並列に動作するので、その状態を忠実に表現するために連続代入文や always プロシージャが使用されます。一方、initial プロシージャはテストベンチで各種の初期化処理に使用されます。

本書のプロセス制御の解説には検証作業で使用される機能も含まれていますが、それらの機能は基本的で SystemVerilog 使用者の誰もが理解しておかなければならない機能です。例えば、fork ブロックの機能は SystemVerilog の必須知識です。また、SystemVerilog では実行中のプロセスのインスタンスを取得する事ができるので、ユーザはプロセスの停止、および再開等の細かなプロセス制御を行なう事ができます。本章では並列処理の根幹を成すプロシージャとプロセスについて解説します。

6.1 概要

SystemVerilog によるモデリングで使用できるプロシージャは、表 6-1 のようにまとめられます。ここで、センシティビティリストとは動作記述が依存する信号のリストを意味します。その概念を 6.2 節で解説します。

表 6-1　モデリングに使用する `always` プロシージャの種類

always の種類	用途と機能
always_comb	• 組み合わせ回路を記述するための always プロシージャのタイプです。 • 記述が組み合わせ回路のルールに違反すると警告メッセージが発行されます。 • センシティビティリストが自動的に生成されます。 • always_comb プロシージャで指定されている左辺の変数には、他のプロシージャで値を設定する事はできません。
always @(*) always @*	• センシティビティリストが自動的に生成するための always プロシージャのタイプです。基本的には、組み合わせ回路を記述する場合に使用します。@(*)を@*とも書きます。 • @(*)は、always_comb と同様にセンシティビティリストを自動生成しますが、サブルーティンの内部で使用されている信号を含みません。結果として、@(*)は不完全なセンシティビティリストを生成する可能性があります。
always_latch	• ラッチを記述するための always プロシージャのタイプです。 • 記述がラッチ生成のルールに違反すると警告メッセージが発行されます。 • センシティビティリストが自動的に生成されます。
always_ff	• 論理合成可能なシーケンシャル回路を記述するための always プロシージャのタイプです。 • イベント制御はただ一つで、先頭で宣言されなければなりません。 • ディレーを指定する事はできません。 • always_ff プロシージャで指定されている左辺の変数には、他のプロシージャで値を設定する事はできません。
汎用 always	• 組み合わせ回路、およびシーケンシャル回路を記述するための汎用プロシージャのタイプです。 • センシティビティリストの指定により、組み合わせ回路、またはシーケンシャル回路を表現する事ができます。

要約すると、`always_comb`、`always @(*)`、`always_latch`、`always_ff` は汎用 `always` に対して記述的な制限を加えた構造的なプロシージャです。この他にも表 6-2 で示すようなプロシージャがありますが、これらのプロシージは何れも一回だけしか実行しないので、回路を記述するために使用する事はできません。これらのプロシージャはテストベンチで使用されます。

表 6-2　一回だけ実行されるプロシージャ

プロシージャ	機能
initial	• シミュレーション開始時にシミュレータにより一回だけ起動されます。プロシージャの最後の命令の実行が終了すると、そのプロシージャは終了します。 • initial プロシージャは一度だけ実行されるため、デザインをモデリングする事はできません。一般に、テストベンチ等でのデータの初期化処理に使用されます。

11　モジュール

モジュールはデザインやテストベンチを記述するための基本要素で、多くの構文をモジュール内で使用する事ができます。しかも、クラスやインターフェースと同様にパラメータを指定してモジュールを汎用化する事ができます。モジュールは、キーワード module で始まり、キーワード endmodule で終了します。それらのキーワードの間に各種の宣言と定義を記述します（図 11-1）。

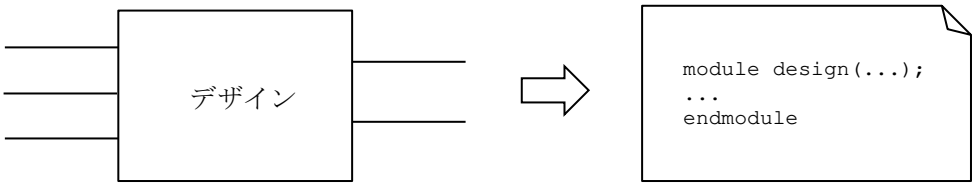

図 11-1　モジュールによるデザインの実装

モジュール内に記述する代表的な例としては以下のような宣言と定義があります。

- モジュール名称
- モジュール記述を汎用化するためのパラメータ宣言
- モジュールの接続仕様を示すポートリスト
- モジュール内の記述で使用するネットや変数の宣言
- 組み合わせ回路を記述するための連続代入文
- 組み合わせ回路やシーケンシャル回路を記述するための always プロシージャ
- テストベンチの初期化設定処理を行うための initial プロシージャ
- 共通処理を定義するためのタスクやファンクション
- ゲートインスタンス
- 他のモジュールのインスタンス

本章ではモジュールに関する知識を解説します。

11.1　シンタックス

モジュールの全体的なシンタックスの概要は、以下のようになります（[1]）。ここで、module_ansi_header が SystemVerilog スタイルで、module_nonansi_header は Verilog スタイルです。本書では、主として、SystemVerilog スタイルのシンタックスを使用します。シンタックスにおいて、module と macromodule は同義語です。

```
module_declaration ::=
    module_nonansi_header [ timeunits_declaration ]
     { module_item } endmodule [ : module_identifier ]
  | module_ansi_header [ timeunits_declaration ]
     { non_port_module_item } endmodule [ : module_identifier ]
module_ansi_header ::=
    { attribute_instance } module_keyword [ lifetime ]
      module_identifier { package_import_declaration }
      [ parameter_port_list ] [ list_of_port_declarations ] ;
module_keyword ::= module | macromodule
```

簡単に言えば、SystemVerilog のモジュールは以下のような構成になります。

- モジュールの定義は、キーワード module とモジュール名称（module_identifier）を指定して始まり、キーワード endmodule で終了します。モジュール名称を endmodule の後にコロンを挟んで指定する事ができます。これはコメントとしての役割をします。
- モジュール内でパッケージを参照する場合、モジュール名称の直後でパッケージをイン

ポートする事ができます。
- モジュールでパラメータを使用する場合、parameter_port_list にパラメータを指定する事ができます。
- モジュールで使用するポートの宣言を list_of_port_declarations に指定します。
- モジュール内部の定義を non_port_module_item に記述します。

これらの内容は次第に明らかになります。最も典型的なモジュール記述例は以下のようになります。

```
module mux2_df(input a,b,s,output out);
assign out = (s == 0) ? a : b;
endmodule : mux2_df
```

この例では、キーワード module に続きモジュール名称 mux2_df が指定され、その後にポート宣言が並びます。モジュール内部が assign 文だけで構成されています。endmodule の後にはコメントとしてモジュール名称が指定されています。対応するシンタックス要素は表 11-1 のようになります。

表 11-1　モジュール mux2_df のシンタックス要素の対応

シンタックス要素	使用例
module_keyword	module
module_identifier	mux2_df
list_of_port_declarations	(input a,b,s,output out)
non_port_module_item	assign out = (s == 0) ? a : b;

次の記述例は parameter_port_list を指定しています。対応するシンタックス要素は表 11-2 のようになります。以下の記述において、キーワード parameter を省略する事もできます。

```
module function_unit #(parameter NBITS=2)
    (input [NBITS-1:0] a,b,wire c2,c1,c0,
    output logic [NBITS-1:0] out);
wire [NBITS-1:0] _or, _nor, _and, _nand, _xor, _xnor;
assign _or = a | b;
assign _nor = ~_or;
assign _and = a & b;
assign _nand = ~_and;
assign _xor = a ^ b;
assign _xnor = a ~^ b;
mux8 #(NBITS)
  MUX8('0,_or,_nand,_xor,_xnor,_and,_nor,'1,c2,c1,c0,out);
endmodule
```

表 11-2　モジュール function_unit のシンタックス要素の対応

シンタックス要素	使用例
module_keyword	module
module_identifier	function_unit
parameter_port_list	#(parameter NBITS=2)
list_of_port_declarations	(input [NBITS-1:0] a,b,wire c2,c1,c0, output logic [NBITS-1:0] out)
non_port_module_item	wire [NBITS-1:0] _or, _nor, _and, _nand, _xor, _xnor; ... mux8 #(NBITS) MUX8('0,_or,_nand, _xor,_xnor,_and,_nor,'1,

12　クラス*

クラスは SystemVerilog の最も重要な機能の一つであるので、本章ではクラスが備える基本的な性質、およびそれらの使用法を解説します。デザインにクラスが使用される事はありませんが、デザインの検証には不可欠な知識です。SystemVerilog のクラスは、Java や C++のクラスよりも豊富な機能を有しています。例えば、乱数発生機能、制約を定義する機能、カバレッジ計算機能、virtual インターフェースの機能等が挙げられます。つまり、SystemVerilog のクラスは、一般のプログラミング言語の持つクラスの機能に加えて検証に特化した機能を備えている複雑な構造体であると言えます。当然のことながら、本章の解説だけではクラスの全てを語り尽くす事はできないので、本章ではクラスが備えている基本的な機能を重点的に解説します。更に進んだ内容を習得するためには巻末の文献を参照して下さい。

12.1　概要

クラスはデータタイプの一つで、データとそれらを操作するサブルーティンから構成されます。クラスのデータとサブルーティンは、それぞれクラスプロパーティおよびクラスメソッドとも呼ばれます。クラスには、プロパーティやメソッド以外を定義する事もできます。例えば、データタイプおよびパラメータをクラス内に定義する事ができます。

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）を作るために使用されます。先ず、クラスがどのような構造体であるかを例により示します。クラス内で宣言または定義された項目の意味は次第に明らかになります。

例 12-1　simple_item クラスの定義例

以下の例は、simple_item と呼ばれるトランザクションのクラスを示しています。このクラスには、プロパーティおよびメソッドの他に、パラメータやデータタイプも定義されています（表 12-1）。extern 宣言されたメソッド print()の内容は、以下に示すようにクラス外に定義されなければなりません。

```
class simple_item #(NBITS=16);                          ┐
typedef enum bit { READ, WRITE } direction_e;           │
static int              counter;                        │
string                  name;                           │
rand bit [NBITS-1:0]    addr;                           │
rand bit [NBITS-1:0]    data;                           │
rand direction_e        direction;                      ├ クラス定義
                                                        │
function new(string name="simple_item");                │
   this.name = name;                                    │
   counter++;                                           │
endfunction                                             │
                                                        │
extern virtual function print();                        │
endclass : simple_item                                  ┘

function void simple_item::print();                     ┐
   $display("name: %s",name);                           │
   $display("addr: %h",addr);                           ├ extern 宣言された
   $display("data: %h",data);                           │ メソッドの定義
   $display("direction: %s",direction.name);            │
endfunction                                             ┘
```

表 12-1　simple_item を構成するメンバー

定義項目	機能または内容
#(NBITS=16)	クラス外部から再定義可能なパラメータです。

direction_e	トランザクションに関する操作の種類を示すenumを定義しています。
counter	トランザクションのオブジェクト数を管理するための変数です。このメンバーは、static属性を伴っています。
name	トランザクションの名称を示します。
addr	NBITSのランダム変数です。
data	
direction	トランザクションに関する操作の種類を示すランダム変数です。
new	コンストラクタです。
print	トランザクションをプリントするファンクションです。

■

参考 12-1

例12-1では、クラスの全容を見易くするために、メソッドprint()の内容をクラス外に定義しています。一般に、クラス内の定義状態を理路整然としておくとクラスの仕様が理解し易くなります。クラス内には多くのメソッドを定義するため、このような配慮が必要になります。ちなみに、メソッドprint()をクラス内に定義すると以下のようになります。

```
class simple_item #(NBITS=16);
typedef enum bit { READ, WRITE } direction_e;
static int              counter;
string                  name;
rand bit [NBITS-1:0]    addr;
rand bit [NBITS-1:0]    data;
rand direction_e        direction;

function new(string name="simple_item");
   this.name = name;
   counter++;
endfunction

function void print();                                    ┐
   $display("name: %s",name);                             │
   $display("addr: %h",addr);                             ├ クラス内に定義した
   $display("data: %h",data);                             │ メソッドprint()
   $display("direction: %s",direction.name);              ┘
endfunction

endclass : simple_item
```

□

12.2 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]）。例 12-1 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]
```

やや複雑なシンタックスとなっていますが、要点をまとめると以下のようになります。

- 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。

したがって、スケジューリング領域の実行順序は非常に複雑になります。時系列的な領域の実行順序を LRM から参照して図 15-2 に示しました。ただし、明確さのために一部を除外してあります。図の右側には、代表的な SystemVerilog 命令が実行する時期を明記してあります。

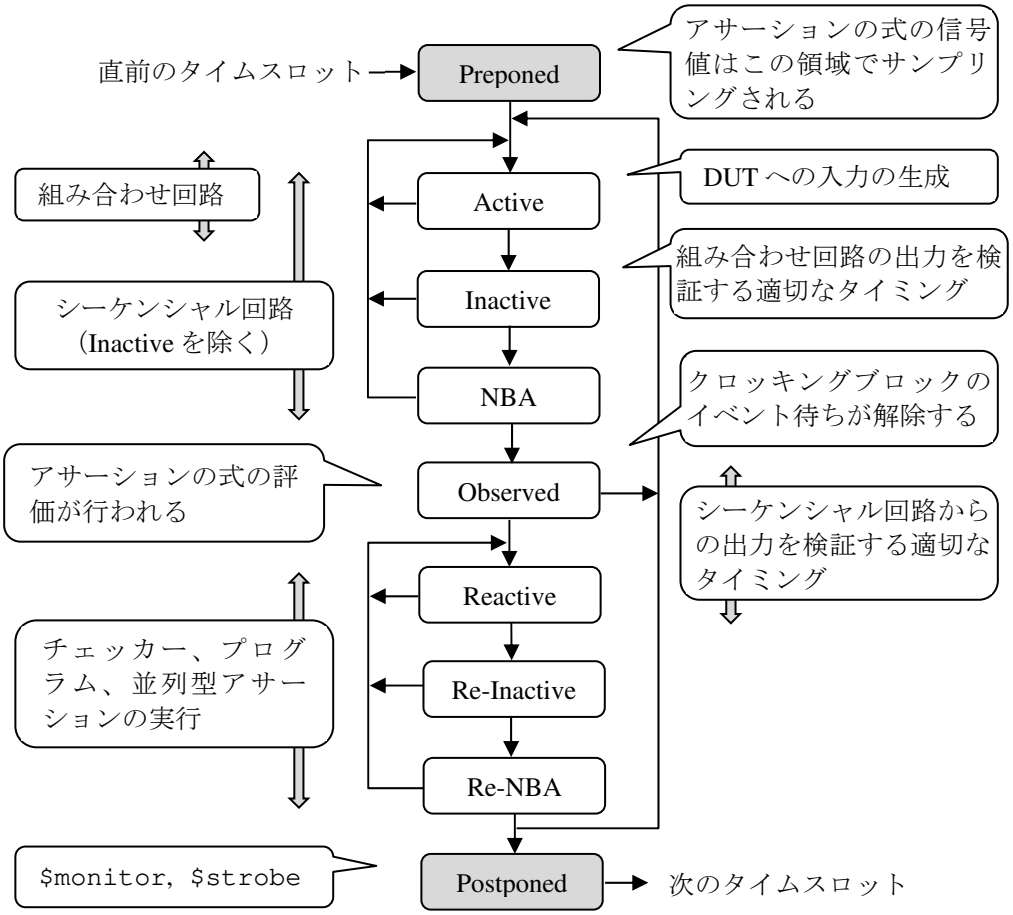

図 15-2　イベントスケジューリング領域（[1]より抜粋）

イベントスケジューリング領域で実行される代表的な動作を表 15-1 にまとめておきます。

表 15-1　イベントスケジューリング領域と代表的な動作の実行

領域	代表的な動作
Preponed	• 時刻 T におけるシミュレーションが開始する最初の領域です。 • アサーションの式で使用されている信号の値はこの領域でサンプリングされます。
Active	• モジュールのブロッキング命令 a = b; 等が実行されます。
Inactive	• モジュールの命令 #0 a = b; 等が実行されます。 • 厳密に言えば、a = b が Inactive 領域にスケジューリングされます。
NBA	• モジュールのノンブロッキング命令 q <= d; 等が実行されます。 • 厳密に言えば、t = d が Active 領域にスケジューリングされ、q = t が NBA 領域にスケジューリングされます。
Observed	• アサーションの評価が実行します。 • クロッキングブロックの待ち状態が解除されます。
Reactive	• プログラムとチェッカーのブロッキング命令が実行します。 • 並列型アサーションの action ブロックが実行します。