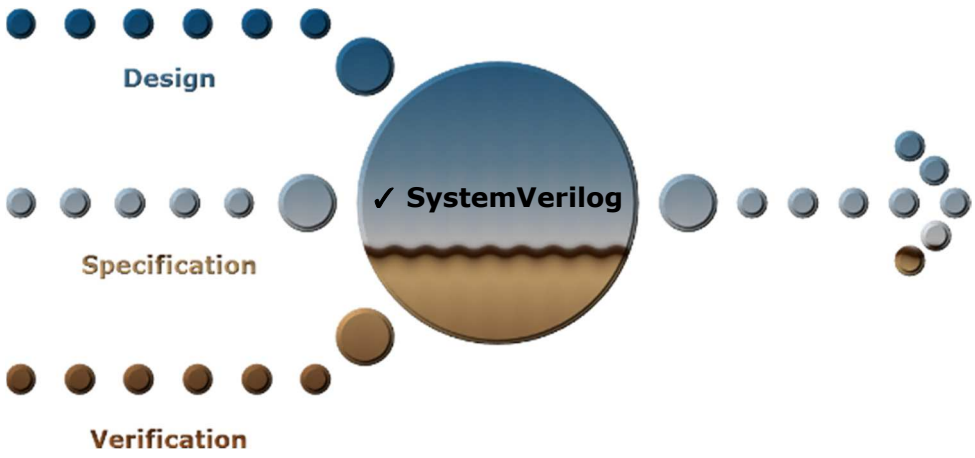
Artgraphics

SystemVerilog 入門

Document Identification Number: ARTG-TD-002-2020
Document Revision: 1.2, 2020.03.31
アートグラフィックス
篠塚一也



SystemVerilog 入門

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

SystemVerilog Primer

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

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。

LRM は多くの研究者、技術者による長年の努力の賜物として完成された 1300 ページにも及ぶ大作です。一方、LRM は標準規格であるが故に、ハードウェアを設計・検証する技術者の誰もが一度は目を通さなければならない言語仕様書です。然し、その必要性にもかかわらず、LRM は容易に理解できる英文で記述されているとは決して言えません。恐らく、言語としての厳密性が記述の厳密性に繋がり、英文としては難解な解説書となっていると思えます。概して、難解な記述は理解し難いだけでなく、読者に依る解釈の差異が生じる可能性が潜んでいるため、結果として、多くの混乱を招きます。SystemVerilog は、機能的に複雑であるだけでなく、言語仕様書としても極めて複雑です。この両者の複雑さが、日本国内における SystemVerilog の実践への適用を妨げている要因の一つであるとも考えられます。本書は、この様な状況を鑑みて、誰もが LRM を誤解なく解釈する事ができる様に基礎知識を提供します。即ち、本書は、SystemVerilog の根幹を成す基本機能、及び難解と考えられる機能を重点的、且つ徹底的に解説し、SystemVerilog を実践に適用する際に必要とされる準備を完全に確立します。

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、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。その意味に於いては、SystemVerilog では従来と異なる発想が求められる事になります。

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、現在では必然的な義務となっています。然し、どの様な良書でも全ての人にとって同様に良書であるとは限りません。それは、人それぞれの思考法、経験、知識、必要性等における差異が、理解度に於ける差として現れるからです。とりわけ、SystemVerilog は欧米人により考え出された言語であり、その概念は多くの点において日本人の思考法とは必ずしも一致しません。例えば、LRM で記述しているインターフェースクラスの概念がそうです。LRM では冗長な解説でインターフェースクラスの概念を解説していますが、日本人的な考えでは、インターフェースクラスは「仕様の標準化」の一言で説明が付きます。寧ろ、その方が日本人にとっては分かり易い事は明らかです。本書は、日本人の思考法に適する様に機能の解説を進めています。

その意味において、本書は単なる SystemVerilog の解説書では無く、言語の持つ意味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 に書かれてある検証機能を除く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 22 章から構成され、SystemVerilog 言語全般の解説をカバーしています。但し、検証機能に関しては概要的な解説になっています。本書は、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、ランダムスティミュラスの生成

を解説しています。第 17 章では、モジュール定義の仕方を解説し、代表的な回路を例にとり SystemVerilog によるモデリング例を紹介しています。これらの例には、組み合わせ回路、シーケンシャル回路、FSM 等が含まれているので、モデリングに関する SystemVerilog 言語機能を使用する知識を再確認する事ができます。しかも、適切な SystemVerilog の機能を使用してそれらのモデリングを検証しているので、検証例は SystemVerilog の機能を例示する最適な素材となっています。要約すると、本書を読了後は SystemVerilog の基礎的な知識を習得する事ができています。紙数制限により、検証機能の詳細、及び最近知られ始めている検証手法 UVM の解説を省略しています。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。割愛した検証機能と UVM に関しては、文献[8]が詳しい解説を含んでいます。

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

変更履歴

日付	Revision	変更点
2020.02.16	1.0	初版。
2020.03.16	1.1	① FSM の事例を変更。
2020.03.31	1.2	① 制約によるランダムスティミュラスの生成の解説を充実。

目次

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。

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];
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。
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。
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。

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。上記の記述にはコンパイルエラーが発行されます。
■

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。n1、n2、n4、n5 は 1 ビットで、n3 は 16 ビットのネットです。下記の n2 の様に、ネット型に対して logic 型を添えても全く効果はありません。

```
wire          n1;
wire logic    n2;
wire [15:0]   n3;
wire #5       n4;
wand          n5;

assign n5 = n1&n2;
assign n5 = ~n3;
```

ネット n4 の様に、ネットの宣言時にディレーを設定する事ができます。このディレーは、ネットディレーと呼ばれます。ネット n4 は、#5 のネットディレーを持つので、n4 のドライバーの値が変化すると、新しい n4 の値は#5 だけのディレーを置いて有効になります。一方、ネット n5 は wand として宣言されているために、n5 が複数のドライバーを持つ場合には、図 3-1 の様に AND ゲートが生成されて、複数ドライバーの値の発生を解消します。

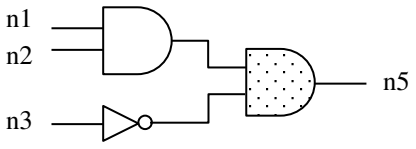


図 3-1　wand の効果

```
@0: waiting for ev to occur.
@10: ev released.
@10: main completed
```

■

3.14 Typedef

typedef を使用すると、既に定義されているデータタイプを用いて新しいデータタイプを定義する事ができます。この方法を用いると既に存在するデータタイプの別名を定義する事もできます。typedef 文は以下の様なシンタックスに従います（[1]）。

```
type_declaration ::=
  typedef data_type type_identifier { variable_dimension } ;
| typedef interface_instance_identifier
    constant_bit_select . type_identifier type_identifier ;
| typedef [ enum | struct | union | class | interface class ]
    type_identifier ;
```

例 3-15　typedef 文の使用例

typedef 文は、複雑な修飾子を伴うデータタイプ宣言を簡略した形式で利用するための手段です。例えば、以下の様に ia10_t を bit signed [31:0]のアレイの代わりに使用する事ができます。

```
typedef bit signed [31:0]     ia10_t [10];

module test;
ia10_t    array;

   initial begin
      foreach(array[i])
         array[i] = i;

      foreach(array[i])
         $display("array[%0d]=%0d",i,array[i]);
   end
endmodule
```

定義したデータタイプ ia10_t を使用している。

変数 array は 32 ビット符号付き整数型のアレイとなります。アレイの大きさは 10 です。アレイの大きさを変更するには、typedef 文の宣言を変えるだけで済むという利点があります。

■

3.15 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。

例 3-16　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。

```
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。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;
```

■

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。

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事ができます。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事ができます。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、データをプロパーティ、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 のクラスの概念は 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); // constructor
   this.name = name;
   addr = a;
   data = d;
   counter++;
endfunction

function string get_name();
   return name;
endfunction

virtual function void print;
   $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、実践に必要となる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 の指定をする事ができます。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、クラス内のプロパーティ、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_ist 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。
⑥ class_item に於いて、クラスのプロパーティ及びメソッドの定義を記述します。

```
    initial $display("      a  b  {co,sum}");
    initial forever @(a,b)
       #0 $display("@%2t: %2d %2d   %2d",$time,a,b,{co,sum});
endmodule
```

このテストベンチを実行すると以下の様な結果を得ます。a < b の関係を満たす様に乱数が割り当てられてテストが実行しています。

```
      a  b  {co,sum}
@10:  9 14   23
@20:  3  4    7
@30:  3 15   18
@40:  9 10   19
@50:  6 15   21
```

■

5.22 インターフェースクラス

タスクやファンクション等を開発する際、開発手法において個人差が出てくるのは当然の事ですが、それらの使用法は統一されていた方が望ましいのは明らかです。仕様の標準化をするために SystemVerilog のインターフェースクラスを利用する事ができます。

インターフェースクラスはルーティン仕様を規格化したクラスで、その規格に準拠する様に他のクラスを定義する仕組みを提供します。類似の機能は、ベースクラスとサブクラスでも実現する事ができますが、両者には実質的に微妙な差があります。先ず、その差から解説を始めます。

5.22.1 概要

多くのクラスに共通する処理がある場合、それらのクラスへのベースクラスを定義し、共通する処理をベースクラスに定義、又は宣言するのは一般的な方法です。例えば、データに対する put()や get()処理が常に必要な場合には、以下の様に、それらの処理をベースクラスに宣言します。

```
virtual class put_get_t #(type DATA=int);
DATA   storage[$];

pure virtual function void put(DATA data);
pure virtual function DATA get();
pure virtual function int empty();
endclass
```

そして、使用するクラス内で、以下の様に、put()と get()の処理を実装します。

```
class fifo_t #(type DATA=int) extends put_get_t #(DATA);

function void put(DATA data);
   storage.push_front(data);
endfunction

function DATA get();
   get = storage.pop_back();
endfunction

function int empty();
   empty = storage.size()==0;
endfunction
endclass
```

然し、put()や get()を呼ぶだけのために、ベースクラスを定義する必要性があるかは疑問です。更に、SystemVerilog ではサブクラスは唯一つのベースクラスしか持てないので、新たに共通

処理が発生すると、ベースクラスへの宣言が増加する現象が起こります。一部のクラスにのみ必要な機能でも、ベースクラスに宣言を追加しなければならない矛盾も出て来ます。SystemVerilog のインターフェースクラスを使用すると、この様な問題を解消する事ができます。この例の場合には、以下の様にインターフェースクラスを定義する事ができます。

```
interface class put_get_ifc #(type DATA=int);
pure virtual function void put(DATA data);
pure virtual function DATA get();
pure virtual function int empty();
endclass
```

こうすると、データ構造を持つベースクラスは、以下の様になります。

```
virtual class put_get_t #(type DATA=int)
                    implements put_get_ifc #(DATA);
DATA   storage[$];
endclass
```

或いは、もはやベースクラスを定義する必要は無いので、前述の fifo_t クラスを以下の様に定義する事ができます。

```
class fifo_t #(type DATA=int) implements put_get_ifc #(DATA);
DATA   storage[$];

function void put(DATA data);
   storage.push_front(data);
endfunction

function DATA get();
   get = storage.pop_back();
endfunction

function int empty();
   empty = storage.size()==0;
endfunction
endclass
```

5.22.2 機能

インターフェースクラスの概念は Java が備えている interface 機能に類似しています。インターフェースクラスは pure virtual メソッド、タイプ宣言、パラメータ宣言のみから構成されるクラスを意味します。但し、以下の制限が適用されます。

- 制約をインターフェースクラスに定義する事はできない。
- インターフェースクラスにカバーグループを定義する事はできない。
- インターフェースクラス内にクラスを定義する事はできない。
- インターフェースクラスを他のクラス内に定義する事はできない。

キーワード extends を使用して、インターフェースクラスは他のインターフェースクラスからメンバーを継承する事ができます。また、キーワード implements を使用してインターフェースクラスから新しいクラスを定義する事ができます。その際、複数個のインターフェースクラスを同時に implements する事ができます。基本的には、インターフェースクラスは、メソッドの呼び出しに関する規則を定める役目を持ちます。即ち、同じインターフェースクラスを使用する事により、メソッドの呼び出し法を標準化する事ができます。

例 5-17　インターフェースクラスの定義と使用例

先ず、インターフェースクラスを以下の様に定義します。インターフェースクラスでは実装すべきメソッドのルールのみを記述します。このインターフェースクラスから継承して定義

12　プロセス間の同期と交信

プロセス間の同期を取るために、SystemVerilog には以下の機能があります。

- セマフォ
- メールボックス
- イベント(@ev、−>ev、ev.triggered()等)

本章ではこれらの機能を解説します。セマフォは共有資源にアクセスする際に使用する排他制御機能です。メールボックスはデータを生成するプロセス（producer）とデータ処理をするプロセス（consumer）を分離し、2 つのプロセスが並列に実行する事ができる様にするための FIFO リストです。

12.1　セマフォ

セマフォは、複数のプロセスが共有資源にアクセスするための排他制御機能です。共有資源には、一定の上限数以内のプロセスだけにアクセスが許されます。上限を超えたプロセスは、空きができるまで、待たされる事になります（図 12-1）。

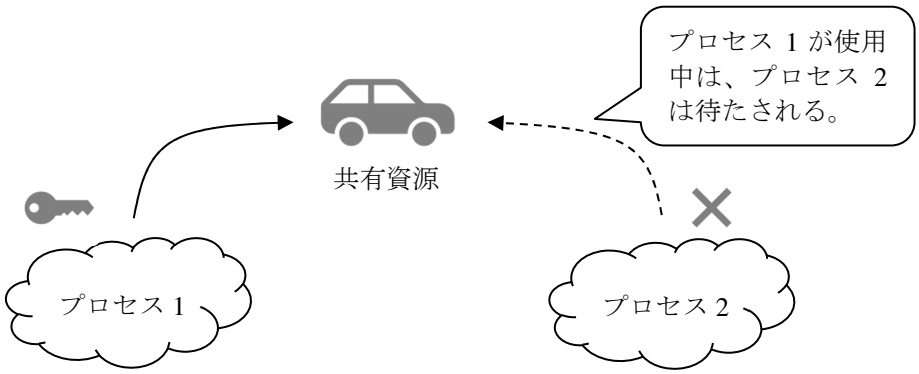


図 12-1　セマフォの概念図

セマフォは SystemVerilog のクラスとして実現されています。クラスハンドルを定義して、ハンドルに 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、

```
semaphore mutex;
mutex = new;
```

の様にして使用します。概念的には、セマフォは一定数のキーを保有するバケットです。利用可能なキーが存在する限り共有資源にアクセスする事ができます。キーが存在しない場合、キーが利用可能になるまでキーを要求しているプロセスは待たなければなりません。

セマフォはロック機能を有するだけのシンプル性があります。共有するデータに関しての情報は一切必要ありません。セマフォから継承して新たなクラスを定義する事も可能です。継承により、メソッド get()を独自に定義する事ができます。セマフォはクラスであるため、表 12-1 の様なメソッドを備えています。

表 12-1　セマフォのメソッド

メソッド	意味
function new(int keyCount = 0);	指定した数のキーを持つセマフォを作成します。バケットは指定されたキーの数で初期化されます。 put で戻すキーの数が get したキーの数よりも

	多い場合、キーの数が keyCount を超える現象が起こります。
function void put(int keyCount = 1);	指定した数のキーをセマフォに戻します。戻すキー数の標準値は 1 です。 もし、利用可能なキーを待っているプロセスに十分な数のキーを戻す場合には、その待ち状態のプロセスの実行が再開します。
task get(int keyCount = 1);	指定した数のキーを取得します。指定数のキーが存在すれば呼び出しているプロセスの実行は継続します。 もし、指定数のキーが存在しない場合には、呼び出しているプロセスの実行がブロックされ、キーの取得が可能になるまで待ち状態になります。
function int try_get(int keyCount = 1);	指定した数のキーを取得します。但し、呼び出しているプロセスをブロックしません。 もし、指定数のキーが存在すれば正の整数を戻します。 もし、指定数のキーが存在しない場合には 0 を戻します。

例 12-1　セマフォの使用例

3 個のキーを持つセマフォの例を以下に示します。

```
module test;
semaphore     lock;
int           rs;

    initial begin
        lock = new(3);

        lock.get(3);
        print(.keyc(3));

        lock.get(2);
        print(.keyc(2));

        rs = lock.try_get(2);
        if( rs )
            print("try_get passed:",2);
        else
            print("try_get failed:",2);

    end

    initial #10 lock.put(2);

function void print(string msg="got",int keyc);
    $display("@%0t: %s %0d keys",$time,msg,keyc);
endfunction
endmodule
```

キーが十分に存在するので、ブロックされずに、get(3)は完了する。

キーを使い切っているので、使用可能になるまで、get(2)は待たされる。

キーを使い切っているので、try_get(2)は 0 を戻して終了する。

$time==10 において、2 つのキーを解除する。

get(3)は直ぐに戻ります。一方、get(2)はキーが存在しないため、$time==10 に put(2)がキーを戻す迄待たされます。get(2)でキーを取得すると、もはやキーは存在しません。このため、try_get(2)は fail します。実行結果は以下の様になります。try_get がブロックせずに戻っている事を確認する事ができます。

但し、p_simple_if はパラメータ化しているので、dev1、及び test は直接 p_simple_if を参照する事はできません。次の様にジェネリックな interface を使用しなければなりません。

```
module dev1(interface intf);  // some device
// ...
endmodule

program test(interface intf); // testbench
// ...
endprogram
```

dev1、及び test は simple_if のビット幅に合わせて展開されます。これは、SystemVerilog コンパイラーの仕事です。

15.6 virtual インターフェース

virtual インターフェースは、インターフェースのインスタンスへのポインターを示します。このポインターをクラスのハンドルと同じ様に使用する事ができます。クラス内にはインターフェースを定義する事ができないので、インターフェースにアクセスするためには virtual インターフェースを使用しなければなりません。図 15-2 はインターフェースのインスタンスと virtual インターフェースの関係を示しています。

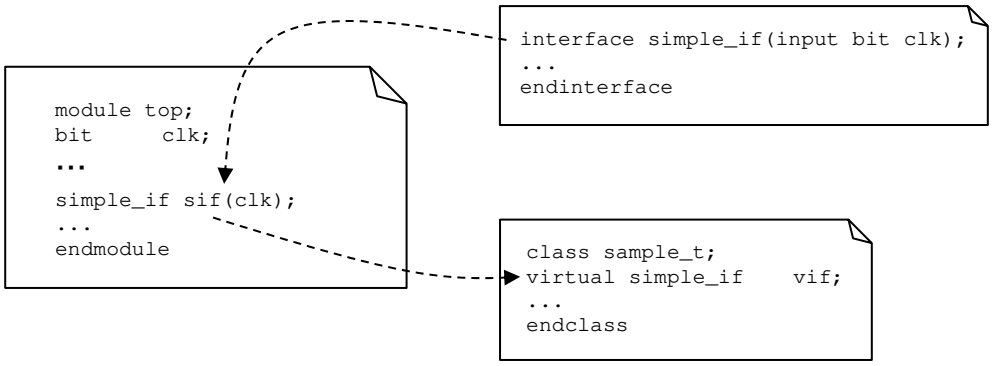


図 15-2　インターフェースのインスタンスと virtual インターフェース

クラス内では virtual インターフェースを以下の様に宣言します。

```
class sample_t;
string name;
virtual simple_if   vif;      // virtual interface

function new(string name);
   this.name = name;
endfunction

function void set_vif(virtual simple_if sif);
   vif = sif;
endfunction

endclass
```

virtual インターフェースは null で初期化されるため、使用するまでにインターフェースのインスタンスで初期化しなければなりません。

例 15-1　virtual インターフェースの使用例

ここでは、クロッキングイベントに同期して信号 a と b をスワップするモジュールの動作を

検証する検証環境を構築します（図 15-3）。

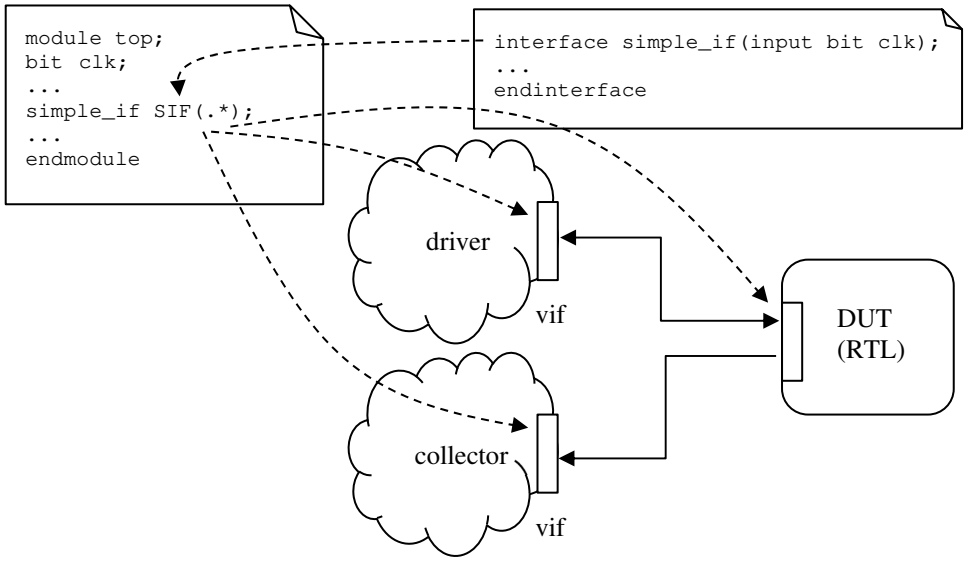


図 15-3　virtual インターフェースを使用する検証環境

最初に、インターフェースを以下の様に定義します。

```
interface simple_if(input bit clk);
logic  a, b;

    clocking cb @(posedge clk); endclocking
    modport DUT(input clk,inout a,b);
    initial begin
        a = 0;
        b = 0;
    end
endinterface
```

検査対象のモジュールは以下の様に定義されます。ポートには、modport を使用します。

```
module dut(simple_if.DUT mp);
    always @(posedge mp.clk) begin
           mp.b <= mp.a;
           mp.a <= mp.b;
    end
endmodule
```

クラスを使用して DUT の動作を確認する事にします。そのために、トランザクションを以下の様に定義します。

```
class data_item_t;
rand bit [15:0]  aux;
bit              a, b;

function void post_randomize();
    a = ^aux;
    b = aux[0];
endfunction
endclass
```

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
```

■

17.7.2 シーケンシャル回路

シーケンシャル回路の例としてバイナリーカウンター、JK―フリップフロップ、Johnson カウンターの記述例を紹介します。記述例を紹介する前に、RTL 論路合成可能なシーケンシャル回路の記述方式を解説しておきます。

17.7.2.1 シーケンシャル回路の記述ルール

always プロシージャを使用して RTL 論理合成可能なシーケンシャル回路を記述する際に守るべきルールは、以下の様になります。

① クロック信号と非同期信号以外は、センシティビティリストに指定しない。
② 非同期信号を条件判定に使用する場合、条件判定とエッジが一致しなければならない。
③ クロック信号は、センシティビティリスト以外に出現してはならない。
④ 非同期信号は、必ず、条件判定に使用されなければならない。
⑤ 一般的に、ノンブロッキング命令（<=）を使用して記述する。

例えば、次の記述は非同期な set、及び reset 信号付のフリップフロップの例です。この記述例は、上記の条件を満たしているので、RTL 論理合成可能なシーケンシャル回路です。

```
module async_flipflop(input logic clk,set,reset,data,
                      output logic q,q_bar);

assign q_bar = ~q;

always @(negedge set or negedge reset or posedge clk) begin
    if( reset == 0 )
        q <= 0;
    else if( set == 0 )
        q <= 1;
    else
        q <= data;
end

endmodule
```

エッジが negedge reset なので
　if(reset == 0)
又は、
　if(!reset)
の様に判定しなければならない。

センシティビティリストには、クロックと非同期信号だけが指定されています。更に、非同期信号のエッジと条件判定も一致しています。そして、クロック信号はセンシティビティリスト以外の記述に使用されていません。最後に、出力信号への値の割り当てにはノンブロッキング命令（<=）を使用しています。

17.7.2.2 バイナリーカウンター

このバイナリーカウンター（図 17-10）には、クリア信号（pc）、ロード信号（load）、アップダウン信号（up）が付いています。load==1'b1 であれば、指定された値（data）をロードします。pc も load もセットされていない場合には、up の状態によりカウンターの内容を更新します。

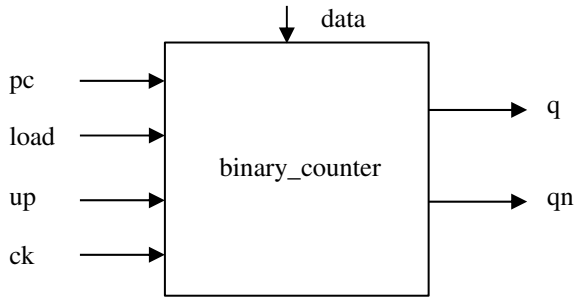


図 17-10　バイナリーカウンターのブロックダイアグラム

例 17-9　バイナリーカウンターの記述例

モジュール binary_counter を以下の様に定義します。

```
module binary_counter #(parameter NBITS=2)
    (input ck,up,pc,load,input [NBITS-1:0] data,
    output [NBITS-1:0] q,qn);
logic [NBITS-1:0]       counter;

assign q = counter;
assign qn = ~counter;

always @(posedge ck)
    if( pc )
        counter <= 0;
    else if( load )
        counter <= data;
    else if( up )
        counter <= counter + 1;
    else
        counter <= counter - 1;

endmodule
```

次に、テストベンチを以下の様に定義します。binary_counter はシーケンシャル回路なので、その出力を確認するタイミングには注意しなければなりません。ここでは、クロッキングブロックで同期を取って binary_counter からのレスポンスをチェックします。

```
module test;
parameter SIZE = 4;
bit       clk, up, pc, load;
logic [SIZE-1:0] d, q, qn;

clocking cb @(posedge clk); endclocking
binary_counter #(.NBITS(SIZE)) DUT(.*,.ck(clk));

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

always @(cb)
    $display("@%3t: pc=%b load=%b up=%b d=%2d q=%2d(%b) qn=%2d(%b)",
```

クロックをゲートで制御すると skew が発生してしまうため、この例ではフリップフロップのデータ入力をゲートで制御する様にします。この場合には、レジスタ出力をフィードバックさせる必要があります。そのためには、else クローズを伴わない不完全な if-else-if 構造を記述しなければなりません。

```
module gated_clock(input clk,reset,data_gate,data,output logic q);

    always @(posedge clk,posedge reset)
       if( reset == 1 )
          q <= 0;
       else if( data_gate )
          q <= data;

endmodule
```

上記の記述には else クローズが無いため、(reset != 1) && (data_gate != 1)の場合には、q には値が設定されないので、q は現在の内容を維持する事が分かります。尚、テストベンチと実行結果は、前例と殆ど同じであるため省略します。

■

17.7.3 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。FSM の状態はレジスタ（フリップフロップ）に保存されます。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。

17.7.3.1 概要

FSM としては、表 17-12 に示す様な二種類のタイプが知られています。

表 17-12　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。 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。

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 17-14 は odd パリティチェッカーを示します。

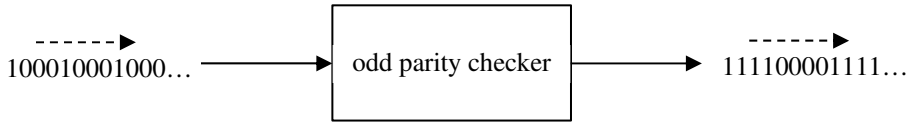


図 17-14　パリティチェッカーのブロックダイアグラム

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。

17.7.3.2 Mealy FSM モデリング

Mealy タイプの FSM は、図 17-15 の様な構成になり、出力は入力と現在の状態に依存します。Mealy FSM のモデリングは以下の様な手順を含みます。

① クロッキングイベントを持つ always プロシージャで状態を更新する。即ち、そのプロシージャにおいて、次の状態を現在の状態に移行する。
② 現在の入力と現在の状態を基にして、組み合わせ回路用の別の always プロシージャで出力と次の状態を計算する。このプロシージャで計算された次の状態は、次のクロッキングイベントが起こるまで反映されない様にする。
③ 現在の状態と次の状態を示す変数を別々に確保する。

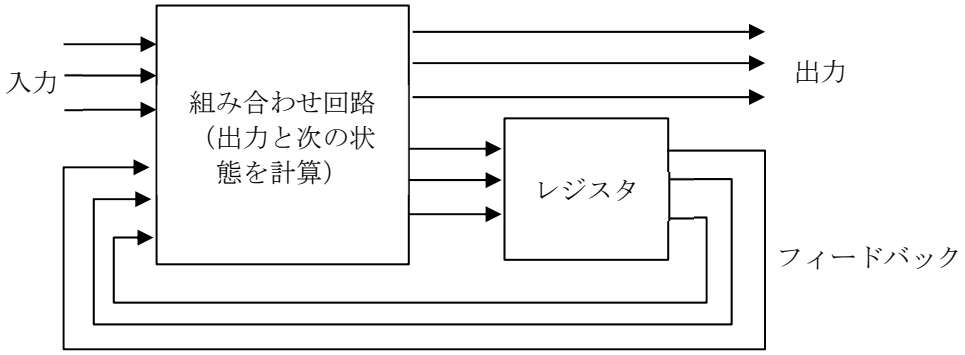


図 17-15　Mealy FSM（[9]）

一般的な Mealy FSM のモデリングは、以下の様な構造を取ります。

```
module mealy_fsm(input clk,reset,...,output logic out);
state_e   state, next_state;

                                  現在の状態と次の状態を示す
                                  変数を宣言する。

   always @(posedge clk,posedge reset)
      if( reset )
         state <= S0;                          状態を更新する。
      else
         state <= next_state;

   always @(state,other_inputs)
      case (state)
      S0: begin
         out = ...;
         next_state = ...;                     現在の状態と現在の
         end                                   入力を基にして、出
      S1:  begin                               力と次の状態を計算
         out = ...;                            する。
         next_state = ...;
         end
      ...
      endcase

endmodule
```

Mealy FSM の出力は、入力に同期するため、クロックと非同期に更新されます。然し、組み