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: 1.0, 2020.11.30
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
Logic Design with Hardware Description Language

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 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、プログラム、チェッカー、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確信しています。

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、検証を専門と志す技術者は、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する事が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 16 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 の使用法に興味が沸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、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、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。それらの内容は、初読時にはスキップ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。

目次

本書で使用する略語一覧

略語	定義
ALU	Arithmetic Logic Unit
BCD	Binary Coded Decimal
BNF	Backus-Naur Form
DDR	Double Data Rate
DMA	Direct Memory Access
DPI	Direct Programming Interface
DUT	Design Under Test、又は、Device Under Test
EDA	Electronic Design Automation
FIFO	First In First Out
FSM	Finite State Machine
HDL	Hardware Description Language
LHS	Left Hand Side
LRM	Language Reference Manual、即ち、IEEE Std 1800-2017
LSB	Least Significant Bit
MOS	Metal-Oxide-Silicon、又は Metal-Oxide-Semiconductor
MSB	Most Significant Bit
OOP	Object Oriented Programming
RTL	Register Transfer Level
SOP	Sum of Products
TLM	Transaction Level Modeling
UDP	User-Defined Primitive

1 概要

1.1 SystemVerilog

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。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 1-1）。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。本章は 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。

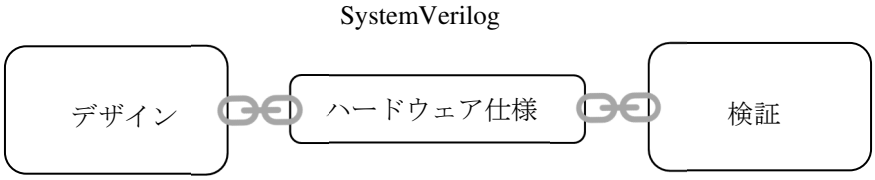


図 1-1　SystemVerilog の機能範囲

1.2 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）で結合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。

1.3 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。このレベルでは、厳密なタイミング検証は行われません。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、信号の値が変化する状態を意味します。SystemVerilog では、単純に信号が変化する状態と特別な値に変化する状態を表現する事ができます。イベントが発生する状態を式 `@(...)` で表現します。

例えば、次の記述は信号 `a` と `b` を入力に持つ `AND` 回路を表現しています。`a` と `b` の何れかが変化すると、オペレーション `a&b` が実行します。これらの信号が変化しない場合にはオペレーションは行われません。したがって、イベントドリブンのシミュレーションの実行効率は良いと考えられます。

[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

```
always @(a,b)
    out = a&b;
```

SystemVerilog では、クロック信号のように特別な値に変化する状態を以下のように記述します。この記述は、非同期リセット信号を持つエッジトリガフリップフロップのデザインを示しています。

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

このようなモデリング記述に対応してターゲットテクノロジーに合致したゲート表現を実現するのが論理合成ソフトウェアツールの役割です。

論理合成ソフトウェアツールは HDL で記述されたデザインからターゲットテクノロジーのライブラリーを使用してゲートレベルネットリストを生成します。例えば、実装する FPGA のテクノロジーに合わせてネットリストを生成する論理合成ソフトウェアツールが用意されています。

1.4 デザインの表現形式

デザインは実装した内容で、SystemVerilog による機能的な記述形式を多く含みます。一方、前述のネットリストは構造的な表現形式であるため、表現自体から機能を理解する事はできません。最も概要的なデザインの表現形式は図 1-2 に示すようなブロックダイアグラム[2]です。SystemVerilog によるデザイン記述は、入力を出力に変換するための手順を詳細に定義します。

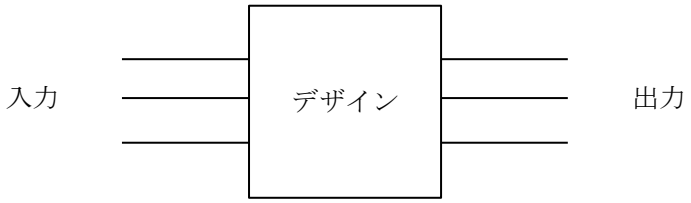


図 1-2　デザインのブロックダイアグラム表現

ブロックダイアグラムにおいては、入力、および出力が明記され、仕様解説は自然言語で補足されます。仕様を SystemVerilog のどのレベルで表現し直すかで、記述の読み易さが大きく変わります。例として、ハーフアダーを仮定します。そのブロックダイアグラムは、図 1-3 のように書く事ができます。

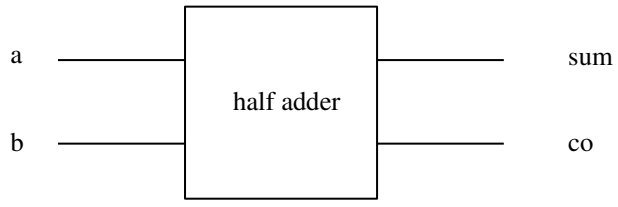


図 1-3　ハーフアダーのブロックダイアグラム

もし、ハーフアダーをゲートレベルで表現すれば、以下のような記述を導く事になります。

[2] 本書のブロックダイアグラムでは、原則として信号は左から右へ流れると仮定します。非標準的な流れの場合には矢印を追加する事にします。

```
module half_adder(input a,b,output co,sum);
xor(sum,a,b);
and(co,a,b);
endmodule
```

しかし、この記述方式はハーフアダーの計算アルゴリズムを理解している事を仮定しています。アルゴリズムに関する知識が無いか、または記憶が定かではない場合には、この記述内容を確認するために少なからずの時間を必要とします。そもそも、HDL はハードウェアの実装に近い記述方式を回避するために存在します。したがって、上記の記述に間違いはありませんが HDL という高レベルの記述能力を十分に活用していないと言えます。もう少し高度な表現法は、以下のようになります。

```
module half_adder(input a,b,output co,sum);
assign sum = a^b;
assign co = a&b;
endmodule
```

この記述法は、もはや、ゲートレベルではないので多少の改善は見られますが、この記述がハーフアダーの計算アルゴリズムを理解している事を仮定している事実には変わりがありません。もし、計算アルゴリズムの記憶に間違いがあれば、後の工程における検証で不可解な現象の解決に時間を要するだけとなります。むしろ、以下のように記述する事が望ましいと言えます。

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

なぜ望ましいかと云えば、この表現方式であれば論理合成ソフトウェアツールが適切なゲートレベルへの変換をしてくれるからです。例えば、XOR や AND 回路を使用せずに、ハーフアダーを表現するライブラリーセルにマップする可能性が高いと言えます。

次に、もう一つの事例として 4 入力のマルチプレクサを紹介します（図 1-4）。

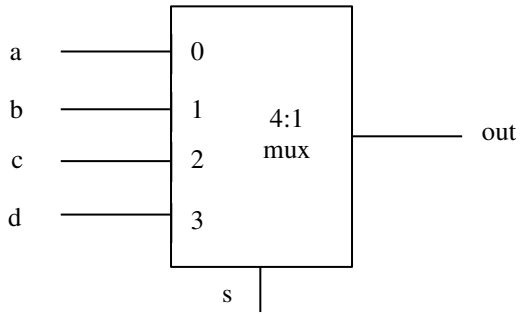


図 1-4　4 入力のマルチプレクサ

4 入力のマルチプレクサをゲートレベルでモデリングすると以下のような記述になります。and、or、not 等の基本的なゲートだけを使用してマルチプレクサの機能を構築する事ができます。

```
module multiplexer_gate(input [1:0] a,b,c,d,[1:0] s,
          output [1:0] out);
wire   _s1, _s0, wa0, wa1, wb0, wb1, wc0, wc1, wd0, wd1;
```

```
not(_s1,s[1]);
not(_s0,s[0]);
and(wa1,_s1,_s0,a[1]);
and(wb1,_s1,s[0],b[1]);
and(wc1,s[1],_s0,c[1]);
and(wd1,s[1],s[0],d[1]);
or(out[1],wa1,wb1,wc1,wd1);
and(wa0,_s1,_s0,a[0]);
and(wb0,_s1,s[0],b[0]);
and(wc0,s[1],_s0,c[0]);
and(wd0,s[1],s[0],d[0]);
or(out[0],wa0,wb0,wc0,wd0);

endmodule
```

確かに、上記の記述は 4 入力のマルチプレクサを実現していますが、これだけ複雑な構造体になると、機能の把握は自明ではありません。仮に、ネットの接続に間違いがあっても容易に発見する事ができないという問題があります。むしろ、以下のように HDL の特性を活用した記述スタイルの方が分かり易いと言えます。

```
module multiplexer_behavior(input [1:0] a,b,c,d,[1:0] s,
       output logic [1:0] out);

always @(a,b,c,d,s)
   case (s)
   0: out = a;
   1: out = b;
   2: out = c;
   3: out = d;
   default out = 'x;
   endcase

endmodule
```

s	out
0	a
1	b
2	c
3	d

この記述によれば、機能は自明と言えます。また、記述に間違いがあったとしても容易に発見する事ができるのは確実です。しかも、賢いコンパイラーは間違いを指摘してくれる可能性さえあります。論理合成ソフトウェアツールは、この記述から `multiplexer_gate` に相当するネットリストを自動的に生成するので、ゲートレベルの記述の必要性はなくなります。

SystemVerilog には、もう一つ別の表現形式があります。その記述法は UDP と呼ばれ、いわゆる真理値表を用いてデザインを表現する方法です。以下は、LRM に紹介されている例です（[1]）。この例は 2 入力のマルチプレクサを記述しています。

```
primitive multiplexer(mux, control, dataA, dataB);
output mux;
input control, dataA, dataB;
   table
      // control dataA dataB mux
      0 1 ? : 1 ;
      0 0 ? : 0 ;
      1 ? 1 : 1 ;
      1 ? 0 : 0 ;
      x 0 0 : 0 ;
      x 1 1 : 1 ;
   endtable
endprimitive
```

例で示すように、SystemVerilog の持つ機能を活用していません。UDP で記述する事ができる

デザインには限りがあるので、本書では UDP を使用したモデリングは割愛します。

1.5　デザインの表現法

本節では、デザインの表現法を解説するために真理値表を利用しますが、従来のようにスタティックな状態で使用するのではなく、論理の最適化の手段として真理値表を利用します。すなわち、真理値表の行の入れ替え、行の削除、列の結合等の手順を経て簡略化された表現を導きます。

仕様を基にしてハードウェアを表現する方法は一意的には定まりません。例えば、2:1 のマルチプレクサを考えてみます。マルチプレクサは、制御信号 `s` の値に従い `a` または `b` の値を出力 `out` につなげます（図 1-5）。

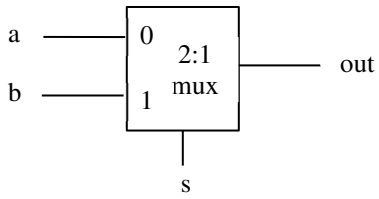


図 1-5　2:1 マルチプレクサのブロックダイアグラム

表 1-1 は 2:1 マルチプレクサの真理値表を示しています。

表 1-1　2:1 のマルチプレクサの真理値表

a	b	s	out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

`out==1` の行だけに着目し書き換えると表 1-3 を得ます。

表 1-2　マルチプレクサの out==1 の行を抽出した真理値表

a	b	s	out
0	1	1	1
1	0	0	1
1	1	0	1
1	1	1	1

さらに、行を入れ替えると表 1-3 を得ます。

表 1-3　簡略化した 2:1 のマルチプレクサの真理値表

a	b	s	out
1	0	0	1
1	1	0	1
0	1	1	1
1	1	1	1

さらに列を結合して書き換えると表 1-4 を得ます。

表 1-4　最適化された 2:1 のマルチプレクサの真理値表

a	b	s	out
1	0	0	1
	1		
0	1	1	1
1			

この表から次の事がわかります。

- s==0 であれば out==a である。
- s==1 であれば out==b である。

したがって、以下のような表現が可能です。ここでは if 文を使用していますが、case 文を使用しても同様な表現が可能です。

```
module mux2(input a,b,s,output logic out);

always @(a,b,s)
    if( s == 0 )
        out = a;
    else
        out = b;

endmodule
```

ちなみに、case 文を使用すると以下のような記述になります。

```
module mux2_case(input a,b,s,output logic out);

always @(a,b,s)
    case (s)
    0: out = a;
    1: out = b;
    default: out = 1'bx;
    endcase

endmodule
```

もちろん、この他の表現法もあります。例えば、以下のようなコンパクトな表現も可能です。

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

さらに、この他にも記述法があります。例えば、以下のような高度な表現法も可能です。

```
module mux2_bitselect(input [1:0] a,s,output out);
assign out = a[s];
endmodule
```

この場合には、s==0 であれば a[0]が、s==1 であれば a[1]が out に設定されます。したがって、この記述は、図 1-6 に示すような回路を意味します。

2　データタイプ

SystemVerilog には、Verilog の持つデータタイプの他に次のようなデータタイプが備えられています。

- `logic`
- `bit`
- `byte`
- `shortint`
- `int`
- `longint`
- `shortreal`
- `string`
- `enum`
- `struct`
- `union`
- `class`

ちなみに、Verilog のデータタイプには以下の種類があります。ただし、`event` は SystemVerilog において機能が拡張されています。

- `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、第 12 章で詳しく解説します。データタイプに関する更なる詳細な解説は、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。データタイプとは、値の集合とその要素に適用されるオペレーションの集合です（表 2-1）。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、または他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　データタイプの例

データタイプ	値の集合	オペレーション
`int`	32 ビットの符号付き整数。	四則演算に加えてビット演算等を適用する事ができる。
`real`	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属された名称付きのエンティティです。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` が真である事を判定しています。`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
```

参考 2-1

`s` が 0、`x`、`z` であると `if( s )` は偽と判定される事に注意して下さい。それを明示的に表現するためには、`if( s == 1'b1 )`、または `if( s == 1 )` のように記述する必要があります。`s` が `x` でも真であると解釈したい場合には、`if( s == 1'b1 || s == 1'bx )` のように表現しなければなりません。
□

参考 2-2

SystemVerilog には 4-state 型と 2-state 型のデータタイプが存在します。4-state 型は表 2-2 で示した 4 つの値を取り得ますが、2-state 型は 0、または 1 だけを取ります。例えば、`logic` は 4-state 型ですが、`bit` は 2-state 型です。実は、値を持たないデータタイプも存在します。例えば、`event` 型のデータタイプは値を持ちません。
□

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。	①　メンバーに名称を付けて管理する事ができる。 ②　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。 ③　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。	①　メンバーには名称が与えられないが、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。 ②　メンバーは同じデータタイプを持つ。 ③　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。 ④　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	① メンバーは連続した領域に配置される。 ② 最初のメンバーが MSB の位置になるように配置される。 ③ 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	① メンバーが連続したメモリーに配置される保証はない。 ② 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を直接定義しているので、anonymous ストラクチャと呼ばれます

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 [3:0] a,b,c,d,e,
    logic s2,s1,s0,output logic [3:0] out);
logic [3:0]  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。

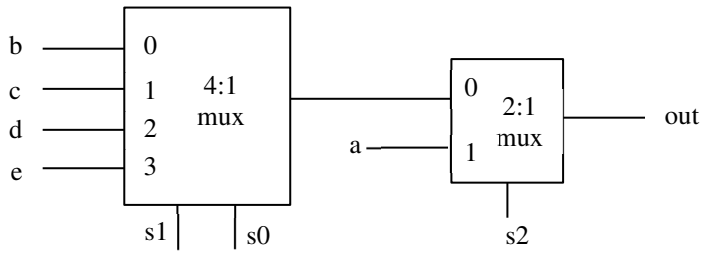


図 5-1　簡単な組み合わせ回路の例

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。

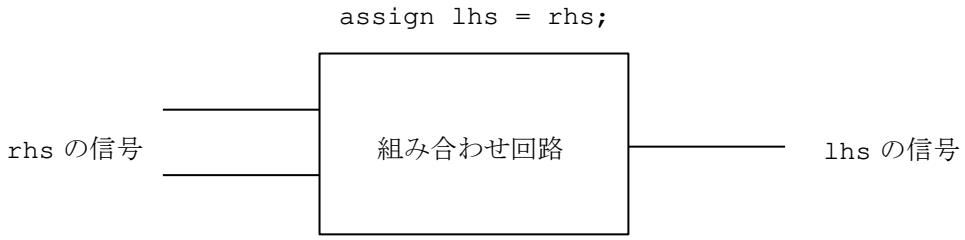


図 5-2　連続代入文の効果

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）。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。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。

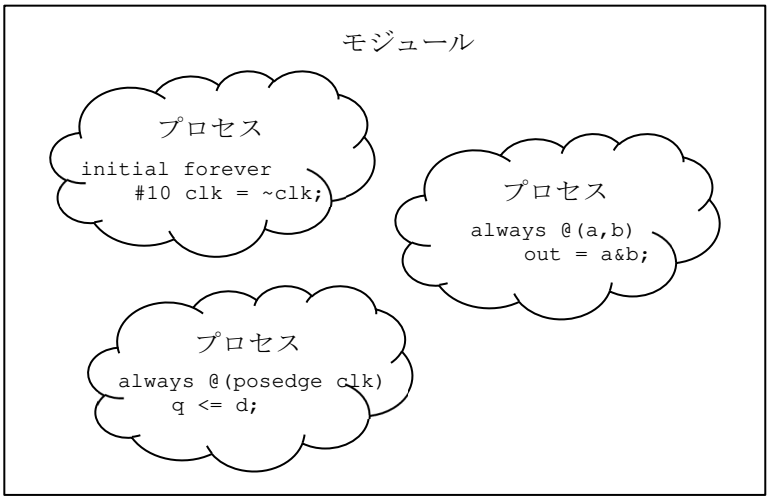


図 6-1　SystemVerilog による標準的なプロセス

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。

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。

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。 ① 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る。 ② センシティビティリストが自動的に生成される。
always @(*) always @*	① 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。 ② @(*)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、サブルーチ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。 ① 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る。 ② センシティビティリストが自動的に生成される。
always_ff	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。 ① イベント制御はただ一つで、先頭で宣言されなければならない。 ② ディレーを指定する事はできない。 ③ always プロシージャで指定されている左辺の変数は、複数のドライバーを持つ事はできない。
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ありません。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。
final	シミュレーション終了後に呼び出されます。シミュレーションが終了しているので、イベントやディレー等の時間を消費する命令を使用する事はできません。通常は、シミュレーション結果を総括する情報をプリントする処理を記述します。

final プロシージャを除く全てのプロシージャはシミュレーション開始と同時に実行を開始して並列に動作しますが、スケジューリングされる順序により実行順序が異なります。例えば、次の記述には四つのプロシージャ p1、p2、p3、p4 が定義されていますが、$time==0 において、どのプロシージャが一番先に実行するかはわかりません。実行順序に依存しないハードウェア記述法が適用されなければなりません。

```
initial begin: p1
...
end

initial begin: p2
...
end

always begin: p3
...
end

always begin: p4
```

7　実行文

実行文はデザインを記述するために使用されます。通常、条件に従い動作を分ける状況が必要になるため、if 文や case 文などの機能が使用されます。本章では、デザインだけでなくテストベンチを記述する際に必要となる機能の解説も含みます。デザインをすると、必ず、検証をしなければならないので、SystemVerilog の実行文に関する正しい知識を持つ必要があります。本章では、以下の文を紹介します。

- if 文
- case 文
- ループ文（for、foreach、repeat、while、do-while、forever）
- return 文
- break 文
- continue 文

SystemVerilog には Verilog に存在しない多くの機能が含まれています。特に、if 文と case 文には RTL 論理合成を支援するための拡張機能として、priority、unique、unique0 が追加されています。これらの追加機能はデザインの検証を強化する効果もあります（表 7-1）。

表 7-1　if 文と case 文の拡張機能

文	priority	unique	unique0
if	① else クローズが指定されずに、if 文で指定した条件が成立しない場合、警告メッセージが発行される。ただし、unique0 の場合には、警告メッセージは抑止される。 ② RTL 論理合成の full case に相当する。		
	① 指定された条件を上から順に走査し最初に一致した条件を採択する。	① if 文に指定された条件はオーバーラップがなく、排他的である事を意味する。したがって、条件判定の順序には依存しない事になる。 ② RTL 論理合成の parallel case に相当する。	
case casex casez	① どの case_item とも一致しない状況が発生すると、警告メッセージが発行される。ただし、unique0 の場合には、警告メッセージは抑止される。 ② RTL 論理合成の full case に相当する。		
	① 指定された case_item を上から順に走査し最初に一致した条件を採択する。	① 指定された case_item にはオーバーラップがなく、排他的である事を意味する。したがって、条件判定の順序には依存しない事になる。 ② RTL 論理合成の parallel case に相当する。	

7.1　if 文

7.1.1　シンタックス

if 文のシンタックスは以下のようになっています（[1]）。unique_priority が Verilog に対する拡張機能です。これらの拡張機能に関しては、表 7-1 を参照して下さい。

```
conditional_statement ::=
    [ unique_priority ] if ( cond_predicate ) statement_or_null
    { else if ( cond_predicate ) statement_or_null }
    [ else statement_or_null ]

unique_priority ::= unique | unique0 | priority
```

通常の if 文は、以下のように使用されます。

```
if (expression) statement;
else if (expression) statement;
else if (expression) statement;
else statement;
```

最後の else クローズを省略すると、不完全 if と呼ばれ、フィードバック信号が生成される可能性があります。組み合わせ回路の記述では、最後の else を省くとラッチが生成される場合が多く見られます。

例 7-1　if 文による 4 入力マルチプレクサの記述例

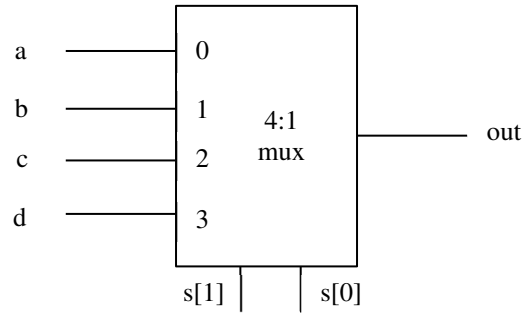


図 7-1　if 文による 4 入力マルチプレクサ

以下の記述は、通常、4 入力マルチプレクサに合成されます（図 7-1）。

```
module mux4(input a,b,c,d,[1:0] s,output logic out);

always @(s,a,b,c,d)
    if( s == 0 )
        out = a;
    else if( s == 1 )
        out = b;
    else if( s == 2 )
        out = c;
    else
        out = d;

endmodule
```

■

7.1.2　unique-if 文

unique-if 文を以下のように使用します。

例 7-2　unique-if の使用例（[1]）

以下に示すように unique を指定します。

```
logic [2:0]  a;

always @(a)
    unique if ((a==0) || (a==1)) $display("0 or 1");
    else if (a == 2) $display("2");
    else if (a == 4) $display("4");
```

unique が指定されているので、次の条件が満たされなければなりません。

9　インターフェース*

SystemVerilog のインターフェースは、設計および検証の両分野で有効に活用する事ができる機能です。一言でいえば、インターフェースはデザインが必要とする信号を一つの集合にまとめる機能ですが、それ以上の効果をもたらす便利な機能です。本章では、インターフェースを設計分野に適用する観点から解説を進めます。

先ず、インターフェースを使用しない簡単なシーケンシャル回路を例にとり解説を進めます。以下のデザインは、リセット信号を持つアップダウンカウンターです。

```
module updown_counter
    (input clk,reset,load,count_up,[3:0] data_in,
    output logic [3:0] count);

always @(posedge clk,posedge reset)
    if( reset )
        count <= 0;
    else if( load )
        count <= data_in;
    else if( count_up )
        count <= count+1;
    else
        count <= count-1;

endmodule
```

このデザインにはそれほど多くのポートが使用されてはいませんが、ポートリストの保守には面倒な作業が付きまといます。例えば、ポート位置を変更する場合、このモジュールを引用しているデザインを確認する必要があります。あるいは、このデザインに新たなポートを追加する場合にも、影響がないかどうかを確認する必要があります。インターフェースを使用すると、このような問題は自動的に解決されます。

9.1　インターフェースの機能概要

SystemVerilog のインターフェースはモジュール間の接続を簡素化するために作られました。機能的には、インターフェースはモジュールのポートを一般化した概念と言えます。

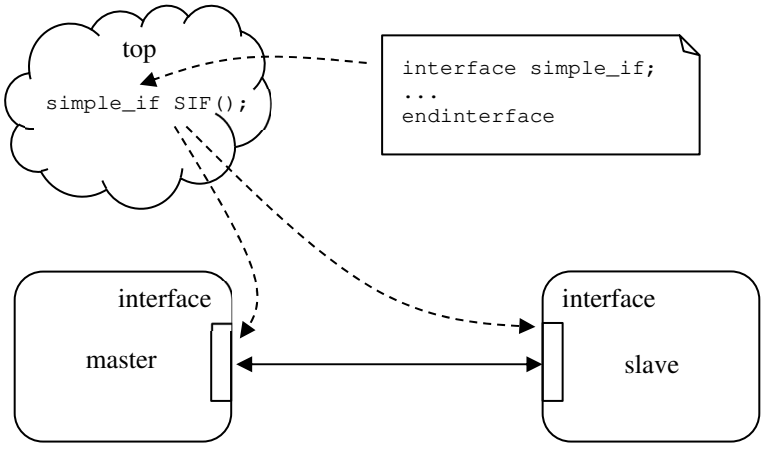


図 9-1　インターフェースによるモジュール間の接続

インターフェースは、モジュールと同様にインスタンスを作る事ができます。そして、インターフェースのインスタンスをポートとして使用し、モジュール間の接続に使用します（図 9-1）。インターフェースには信号を含む事ができます。しかも、それらの信号の向き（input、inout、 output 等）をモジュール毎に変える事ができます。この機能は

modport により実現されます。インターフェースを使用すると、モジュール間の接続変更はインターフェース内の定義変更だけで済むので、それぞれのモジュールのポートリストを変更する必要はなくなります。

インターフェースには initial、および always プロシージャを記述する事ができます。これらのプロシージャを利用すると、信号値の初期化、およびプロトコルのチェック等を容易に行なう事ができます。以下はインターフェースの定義例です。この例は LRM にある記述を多少変更してあります。

```
interface simple_if(input logic clk);
logic        req, gnt;
logic [7:0]  addr, data;

clocking cb @(posedge clk);
input  gnt;
output req, addr;
inout  data;
endclocking

modport DUT (input clk,req,addr,output gnt,inout data);
modport TEST (clocking cb);

endinterface
```

DUT 側の宣言は以下のように modport を使用します。ポートに関する追加、および変更はインターフェース simple_if 内の変更で済みます。

```
module dev1(simple_if.DUT mp);    // some device
// ...
endmodule
```

次の例では、テストベンチにはプログラム[6]を使用しています。記述例にあるように、インターフェース内の信号、クロッキングブロック、modport 等を使用するためには、ドット（.）を使用します。

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

initial begin
   mp.cb.req <= 1;
   wait( mp.cb.gnt == 1 );
   // ...
   mp.cb.req <= 0;
end

endprogram
```

トップモジュールは以下のようになります。

```
module top;
logic  clk;

simple_if SIF(clk);
test TEST(SIF);
dev1 DUT(SIF);
```

[6] プログラムはテストベンチを記述する手段なので、本書ではプログラムの解説を省略します。

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。

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ティおよびクラスメソッドと呼ばれます。クラスには、プロパティやメソッド以外の属性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）を作る時に使用されます。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

以下は、simple_item と呼ばれるトランザクションの定義を示しています。このクラスには、プロパ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、以下の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。

■

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]）。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]
```

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。

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 または automatic の何れかです。省略すると automatic が仮定されます。すなわち、クラス内のプロパーティ、およびメソッドは原則として automatic です。
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。すなわち、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) にお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 において、クラスのプロパーティおよびメソッドの定義を記述します。

多少複雑ですが、これらの機能は次第に明らかになります。例 12-1 で紹介した simple_item クラスに対するシンタックス要素の意味は表 12-2 のようになります。

表 12-2 シンタックス要素の対応

シンタックス要素	使用例
class_identifier	simple_item
parameter_port_list	#(NBITS=16)
class_type	該当せず。
class_item	typedef enum bit { READ, WRITE } direction_e; static int counter; string name; rand bit [NBITS-1:0] addr;

15　シミュレーション実行モデル

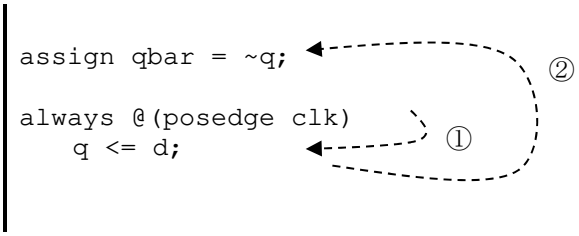
SystemVerilog は並列処理言語であるため、時刻 T におけるプロセス（あるいはスレッド）間の実行順序が明確に定義されなくてはなりません。ハードウェアをモデルした記述の実行が規定された順に実行されなければハードウェアを正確に表現する事はできません。本章では、SystemVerilog が定めているシミュレーション実行モデルについて解説します。本章の内容は、SystemVerilog を理解する上で最も難解でかつ重要な事項です。

15.1　スケジューリング領域

時刻 T におけるシミュレーションが開始されると、プロセスは時系列的に分類された領域で実行します。ある領域のプロセスは他の領域のプロセスよりも先に実行します。この実行順序制約により、状態が安定した信号値を基にしてシミュレーションの論理が確立します。

例えば、`a = b;` が実行する領域は `q <= d;` が実行する領域よりも先に実行します。その他の例として、プログラムで記述されたテストベンチの論理は DUT の信号値が安定した状態になってから実行します。つまり、プログラムの論理は Reactive 領域で実行します。

しかし、記述された命令からだけでは実行順序を決定する事はできません。例えば、以下のような記述例を仮定します。



この場合、時刻 T において `(posedge clk)` のイベントが発生すると、NBA 領域で `q <= d;` が実行した後に q の信号値が更新されて、 `qbar = ~q` が Active 領域で実行します。つまり、同じ時刻 T において、NBA 領域が実行した結果として他の信号値のイベントが発生して Active 領域の命令が誘発されます（図 15-1）。Active 領域の `qbar = ~q` が終了すると、NBA 領域に進みますが、スケジューリングされている命令は既に終了しているので、NBA 領域の次の領域に進行します。

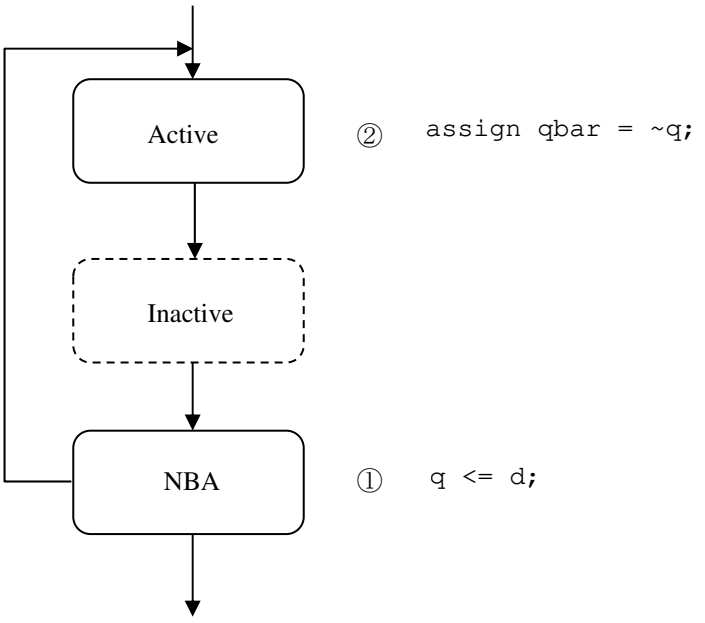


図 15-1　スケジューリング領域の相互関連

例えば、4 ビットの符号付き変数 `offset` は、-8 から 7 までの整数しか表現する事ができません。`offset` の MSB は符号を示し、値を示すためには使用する事ができません。7 に 1 を加えると 8 になりますが、4 ビットの符号付き整数としての解釈では-8 となります。したがい、`offset` に加算を施した後に計算結果にオーバフローが発生すると `offset` の内容は正しい結果を反映しなくなります。本章では、正しいオーバフローの処理法および正しい値の参照法を解説します。

16.3.1 整数表現

符号付き整数は 2 の補数表現で表現されます。二進数の MSB が符号を示し、MSB==1 であれば二進数が負の値を示します。したがって、MSB は符号を示すために使用されるので、値の一部ではありません。MSB を値の一部として使用する場合には、二進数を符号なしとして宣言しなければなりません。表 16-2 は、4 ビットの二進数を例にして、符号により異なった値を意味する事を示しています。図 16-2 は表 16-2 を図として表現しています。

表 16-2　二進数と符号の意味

二進表現	整数	
	符号なし	符号付き
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

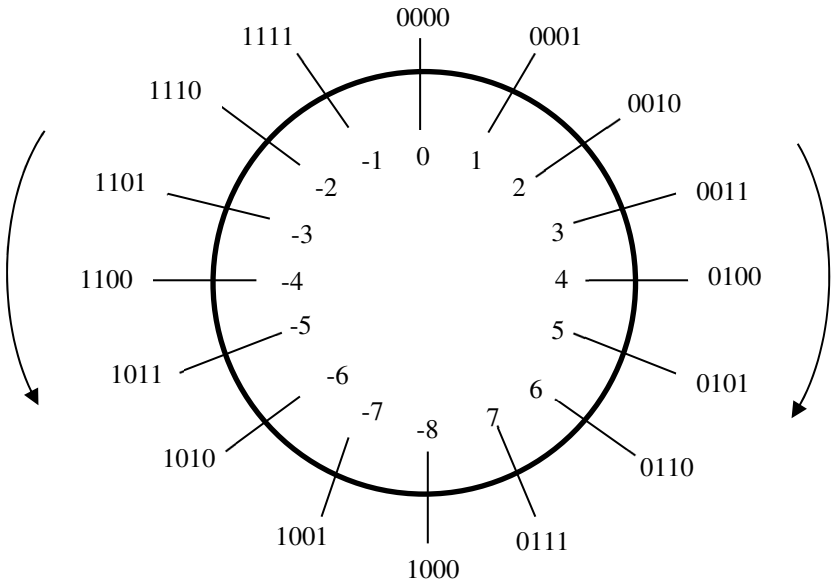


図 16-2　4 ビット符号なし整数と符号付き整数の関係（[2]）