Artgraphics

SystemVerilog 雑談

Document Revision: 7.5, 2025.06.30

アートグラフィックス
篠塚一也

SystemVerilog 雑談

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

SystemVerilog Monologue

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

はじめに

本資料は、書物の素材としては余りにも些細な情報、興味を持つ人がいるかどうかわからない情報、または書物には書ききれない程の詳細な解説から構成されています。また、題材は、筆者が思いついたままに書き並べたもので、必ずしも SystemVerilog LRM の解説順序に準拠している訳ではありません。更に、本資料は SystemVerilog の入門書ではないので、用語の説明や機能の解説等を含んでいません。また、古くなった記事は順次削除して行きます。

本資料は、「SystemVerilog 雑談」と称されていますが、内容的には SystemVerilog LRM の重要な機能の解説、誤解し易い機能に関する補足的な説明、および見落としやすい機能の解説に焦点を当てた意味のある資料で、単なる雑談ではありません。また、本資料は市販されている書物には書ききれない情報を補足する意味も持ちます。

お茶を飲みながら、気楽に読んで下さい。

アートグラフィックス
篠塚一也

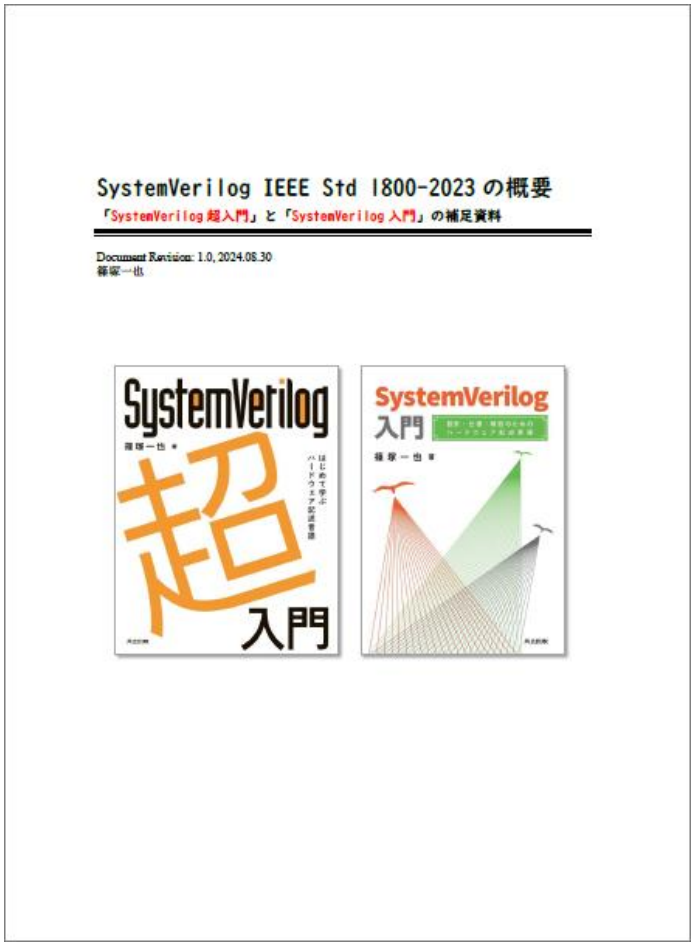
目次

1　IEEE Std 1800-2023　2024.08.30

SystemVerilog が改訂され IEEE Std 1800-2023 として公開されたので、追加された仕様の概要を公開しました。共立出版の『SystemVerilog 超入門』または『SystemVerilog 入門』の書籍ウェブサイトより無償でダウンロードできます。

下記のいずれかのリンクでダウンロードできます。サイトに行き「関連情報タブ」をクリックし、「補足資料」をクリックすると資料を得られます。

https://www.kyoritsu-pub.co.jp/book/b10003280.html
https://www.kyoritsu-pub.co.jp/book/b10031708.html



2　非推奨機能　2022.04.18

SystemVerilog には、使用しない方が良い機能もあります。

SystemVerilog 豆知識

機能	仕様と注意点
defparam	• この機能は Verilog 時代から存在しますが、SystemVerilog の現仕様（IEEE Std 1800-2017）では非推奨になっています。 • defparam は多くの問題を引き起こすとともに、代替機能が SystemVerilog に存在するため、defparam を使用する必要はありません。 • defparam は将来の SystemVerilog 仕様から削除される可能性があるので、使用しない事を勧めます。
procedural assign/deassign	• 現仕様では、defparam と同様に非推奨になっています。したがって、将来の SystemVerilog 仕様から削除される可能性があるので、使用しない事を勧めます。
operator overloading	• 旧仕様（IEEE Std 1800-2012）にはオペレータオーバーローディングの仕様が記載されていましたが、現仕様では、その機能は非推奨になっているので注意下さい。

3　ファンクショナルカバレッジ　2022.09.16

SystemVerilog のファンクショナルカバレッジでは、制約の付いたランダム変数のカバレッジ計算をする場合、状況に応じたカバレッジビンの定義をすると良いです。

制約の付いたランダム変数のカバレッジ計算をする場合、明示的にカバレッジビンの定義が必要になります。例えば、以下の記述例では、カバーポイント a にビン定義がないため auto[0]~auto[7]の 8 個のビンが自動生成されますが、auto[6]と auto[7]はカバーされないので、最大 75%のカバレッジしか達成できません。

```
class sample_t;
rand bit [2:0]   a;

constraint C { a inside { [0:5] }; }

covergroup cg;
    coverpoint a;
endgroup

function new;
    cg = new;
endfunction
endclass
```

ランダム変数 a には制約が定義されている

カバーポイント a にはカバレッジビンが定義されていないので、100%カバレッジを達成できない

カバレッジビンを定義するには、以下のようなオーソドックスな方法があります。

```
covergroup cg;
    coverpoint a { bins value[] = { [0:5] }; }
endgroup
```

これは必要な値を基にした記述法ですが、以下は不必要な値をベースにする方法です。この場合、auto[0]~auto[5]の 6 個のビンしか定義されません。

```
covergroup cg;
    coverpoint a { ignore_bins skip_vals = { 6, 7 }; }
endgroup
```

もし不必要な値にはエラーを報告したい場合には、以下のようにもできます。

```
covergroup cg;
    coverpoint a { illegal_bins invalid_vals = { 6, 7 }; }
endgroup
```

この場合にも、auto[0]~auto[5]の 6 個のビンしか定義されません。何れの方法でも、100%カバレッジを達成する能力を持ちます。

参考文献

SystemVerilog LRM、553-590 ページ
SystemVerilog による検証の基礎、第 4 章、森北出版 2020.

4　カバーポイント　2022.11.11

【初心者向け】SystemVerilog のオペレーションの演算精度は左辺を含むオペランドの精度で決定されるので、十分な計算精度で結果を得られます。然し、式のカバレッジ計算には左辺が存在しないため計算精度の間違いに陥り易い傾向があります。

例えば、以下の記述例において a と b は 1 ビットですが、左辺が 2 ビットであるため、加算は 2 ビットで行われます。

```
logic  a, b, co, sum;
assign {co,sum} = a+b;
```

a と b は 1 ビットであるが、加算は 2 ビットで行われる

然し、以下のカバーポイント(a+b)は左辺を持たないため、演算精度は 1 ビットです。したがって、(a+b)が 0 と 1 になる場合しか情報の収集がされません。つまり、auto[0]と auto[1]の二つのビンしか生成されません。本来、(a+b)は 0~2 の値を取るので三つのビンが必要なので、この計算式の記述は正しくないと言えます。

```
class simple_item_t;
rand logic    a, b;
bit           coverage_enabled;
covergroup cg;
   a_plus_b: coverpoint (a+b) iff (coverage_enabled);
endgroup
...
endclass
```

1 ビットの精度で計算されるので正しくない

このような落とし穴は SystemVerilog の至る所（例えば、アレイ要素の和を求める sum メソッド等）に存在しますが、特に、ファンクショナルカバレッジとアサーションには仕様上、及び使用上の多くの間違い易い機能があります。ちなみに、上記の問題点を以下のように解消できます。他の方法も試して下さい。

```
covergroup cg;
   a_plus_b: coverpoint (a+b+2'b0) iff (coverage_enabled)
   { bins fc_a_plus_b[] = {[0:2]}; }
endgroup
```

参考文献

SystemVerilog による検証の基礎、森北出版 2020.
SystemVerilog 入門、共立出版 2020.

5　posedge と negedge　2023.06.09

SystemVerilog は複雑ですが難しい言語ではありません。ただ、Verilog に比べてより厳密な言語であるだけです。例えば、SystemVerilog では、posedge と negedge のイベントが起こるケースを厳密に定義しています。

posedge と negedge

LSB の値	posedge clk	negedge clk
1 ↕ x, z 0	clk の LSB が以下の変化をすれば posedge イベントが発生します。 0→1 0→x 0→z x→1 z→1	clk の LSB が以下の変化をすれば negedge イベントが発生します。 1→0 1→x 1→z x→0 z→0
要約	LSB が 1 に向かって変化すれば、posedge イベントが発生します。	LSB が 0 に向かって変化すれば、negedge イベントが発生します。

参考
posedge および negedge のイベントは信号値の LSB で判定されるので、1 ビットの信号を使用しないと正しいイベント制御はできません。

6　SystemVerilog 超入門　2023.06.27

『SystemVerilog 超入門』は、初心者が先輩に追いつくための最強の入門書です。恐らく、先輩は超入門しないので絶好の機会です。超入門は持ち運びやすい大きさで、読み易い体裁になっています。

初心者用

先輩用

7　ショートサーキット評価　2023.07.01

条件式の判定法としてショートサーキット評価がありますが、SystemVerilog と Verilog-2001 の間に微妙な差があります。

SystemVerilog と Verilog では基本的なオペレータにおいて微妙な機能差があります。要点は以下のようになります。

オペレータ（**&&**、**||**）は SystemVerilog でも Verilog でもショートサーキット評価されますが、オペレータ（**&**、**|**）の評価法は異なります。

オペレータ（&、|）の評価法の違いを下表にまとめておきます。

記述 （何れも 1 ビットの変数）	SystemVerilog	Verilog-2001
`result =` `  regA & (regB | regC);`	たとえ、`regA` が 0 でも `(regB | regC)` を評価し続けて結論を出します。	もし `regA` が 0 であれば、`(regB | regC)` を評価せずに `result` は 0 と判定されます。
	ショートサーキット評価しない。	ショートサーキット評価する。

したがって、例えば、評価式の中にファンクションの呼び出しが含まれていれば、副作用が出ると SystemVerilog と Verilog では動作が異なる原因となります。SystemVerilog を使用している限りは問題ありませんが、Verilog から SystemVerilog に移行する場合には注意が必要です。

SystemVerilog のショートサーキット評価に関しては以下の文献に解説があります。

SystemVerilog 超入門、共立出版 2023.

8 Inside 2024.01.19

【初心者向け】SystemVerilog には便利なオペレータが多くあります。とりわけ、inside オペレータは省力化の効果があるので、範囲指定がある場合には必須のツールになります。

例えば、下記の check_name()メソッドは名称のチェックをしています。

```
function int check_name(string name);
        check_name = name.len && (name[0] inside {["a":"z"],["A":"Z"],"_"});
        if( check_name )
                for( int i = 1; i < name.len; i++ ) begin
                        check_name = name[i] inside
                                {["a":"z"],["A":"Z"],["0":"9"],"_"};
                        if( !check_name )
                                break;
                end
endfunction
```

ここで用いた名称のルールは以下のようになっています。

- 名称は英数字とアンダースコアで構成されます。
- 名称は英文字またはアンダースコアで始まらなければなりません。

inside を使用しないと冗長になると共に論理が分かり難くなります。

9　キュー　2024.05.09

SystemVerilog のキューに対しては、$を利用すると push_back()を使わなくても済みます。

SystemVerilog では、LHS として q[$+1]を使用すると、新しくキューの要素を追加する事を意味します。例えば、以下のように使用できます。

```
int    q[$];

for( int i = 1; i <= 5; i++ )
    q[$+1] = 2*i+1;
```

結果として、q は {3,5,7,9,11}となります。以下のようにもできますが、中間的にキューが作成されるので効率は良くありません。

```
for( int i = 1; i <= 5; i++ )
    q = {q,i*2+1};
```

10　制約　2024.06.04

SystemVerilog のクラスに制約を定義する場合、機能の使用法によっては歴然とした差が出てきます。特に、ランダムアレイの制約には慎重さが求められます。

簡単な例を使用して、制約の定義法が重要な役割をする事実を紹介します。ここでは、以下のようなランダムアレイを定義して、ソートする事を考えます。

```
rand byte unsigned  d[];
```

以下に、二通りのソート法を準備しますが、一般的に記述法 2 の方が高速です。

記述法 1	記述法 2

class sample_t;
rand byte unsigned d[];
 	
constraint C_D {
　d.size inside {[50:100]};
　foreach(d[i])
　　i < d.size-1 -> d[i] < d[i+1];
}
endclass
	class sample_t;
	rand byte unsigned d[];
	
	constraint C_D {
	　d.size inside {[50:100]};
	　unique{d};
	}
	function void post_randomize();
	　d.sort();
	endfunction
	endclass

一般的に処理時間が多くかかります。	直ぐに処理が完了します。

11　規定幅を持つ整数系　2024.06.25

SystemVerilog には、byte, shortint, int, longint, integer, time 等の規定幅を持つ整数系データタイプが定義されていますが、これらのデータタイプに対して packed 次元を指定する事はできません。

LRM から SystemVerilog のルールを以下に引用しておきます。

IEEE Std 1800-2023 on page 153

Integer types with predefined widths shall not have packed array dimensions declared. These types are byte, shortint, int, longint, integer, and time. Although an integer type with a predefined width n is not a packed array, it matches (see 6.22), and can be selected from as if it were, a packed array type with a single dimension [n-1:0].

したがって、以下のような定義はエラーになります。

```
int [31:0]        value;     // ILLEGAL
```

たとえ、このような記述を許すコンパイラーがあったとしても、それらは IEEE の SystemVerilog 規準を満たしていないので、正しくありません。しかし、以下のように定義されたかのように扱えます。

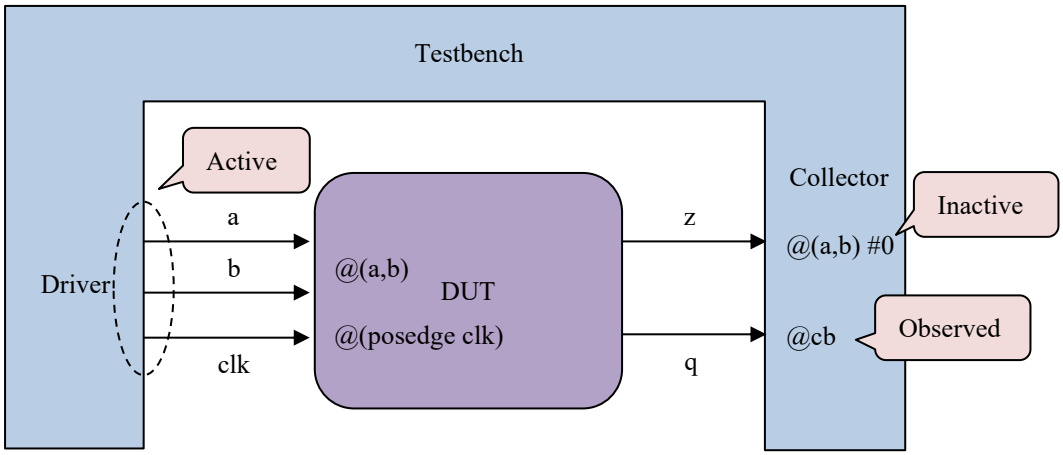
標準的な宣言	マッチする表現
`int    value;`	`bit signed [31:0] value;`

12　SystemVerilog シミュレーション原理　2024.07.01

SystemVerilog は、厳密なシミュレーション原理に基づいて構成されているので、慣れるまでは、標準的な手法を用いて、間違いのない確実な検証手順を確立すると良いです。

例えば、以下のようにすると DUT からの出力信号の値が落ち着いた時点で結果を検証できます。

- 入力を Active 領域で生成して、DUT をドライブする。
- 組み合わせ回路の出力は Inactive 領域で確認する。
- シーケンシャル回路の出力は Observed 領域で確認する。



clk：クロック信号
a, b：組み合わせ回路の入力
z：組み合わせ回路の出力
q：シーケンシャル回路の出力

clocking cb @(posedge clk); endclocking

13　連鎖　2024.07.03

SystemVerilog では、メソッドの呼び出しの連鎖が可能です。例えば、state が enum 型の変数であるとすると、state.name().len のような呼び出しが可能です。

14　String　2024.07.07

些細な事ですが、SystemVerilog では、複数の文字列の繰り返しを指定できます。シンタックスは、{multiplier{Str1,Str2,...,Strn}}のようになります。

15　default　2024.07.10

SystemVerilog には便利なキーワード default がありますが、機能・用途は様々です。

SystemVerilog のキーワード default の用途の代表例

使用例	意味
string 　words[int] = '{**default**:"hello"};	標準値の指定をする。
logic [3:0] a[5]; a = '{**default**:2};	
casex({sa,sb,sc}) 3'b1xx: out = a; 3'b01x: out = b; 3'b001: out = c; **default** out = d; endcase	入力信号値が範囲外である場合の標準的な動作を指定する。

class sub1_t extends base_t;
byte m_value;
function new(**default**,byte v);
　super.new(**default**);
　m_value = v;
endfunction
endclass
	この場合の default は標準値の意味ではなく、ベースクラスのコンストラクタの引数リストを総称する意味を持ちます。

x dist {[100:102]:/3,**default**:/1};	x は様々な値を取り得ますが、区間[100:102]には重さ 3 が割り当てられ、それ以外の数値全体には、重さ1が割り当てられる事を意味します。つまり、x がどのような値をとっても式は真となります。 しかし、default を取り除くと、x が[100:102]以外では偽となります。この場合には、case/casex/casez 文の default に似ています。

16　クラス　2024.08.14

UVM のように、他の企業・団体・組織で開発されたパッケージを使用して、検証システムを構築している場合、次のバージョンにおけるパッケージの変更に備えての対策を予め確立しておく必要があります。

幸い、SystemVerilog にはパッケージの変更に備えて妥当な対策を実現する機能が備わっています。以下に、一例を紹介します。

```
class simple_driver_t extends uvm_driver #(simple_item_t);
...
function new(default);
    super.new(default);
endfunction
...
extern function :extends void build_phase(uvm_phase phase);
extern task :initial get_and_drive();
endclass
```

キーワード	対処の解説
default	クラスのコンストラクタでは、キーワード **default** を使用しておけば、殆どの変更に耐えられます。全く変更なしか、または僅かな変更で済みます。
:extend	こうしておくと、UVM が改訂されて build_phase()の仕様に変更があるとコンパイラーが自動的にエラーを発行してくれます。もし**:extends** が無いと、build_phase()の仕様が異なってもエラーが出ません。しかも、仕様が異なる場合、このクラスに定義されている build_phase()は virtual メソッドではなくなり、検証システムは正しく動作しません。したがって、問題点の解決に時間を要する事になります。
:initial	get_and_drive()は UVM に定義されていない事を仮定しているので、**:initial** を付けています。このメソッドが UVM に virtual メソッドとして定義されていれば、コンパイラーがエラーを発行するので安全です。

この他にも適用できる各種の機能があるので、技術者自身の探求が必要になります。

参考文献

Kazuya Shinozuka, “A Subjective Review on IEEE Std 1800-2023,” DVCon Japan 2024.

17　Verilog　2024.08.15

猛暑日が続いていますが、Verilog から SystemVerilog に移るかどうか迷っている方も多いと思います。ここでは、Verilog と SystemVerilog の相違を簡単な例で紹介します。

Verilog で記述すると、ほぼ確実に古式なスタイルとなります。Verilog には十分な記述能力がないだけでなく、型にはまった記述法が取られる事が主な原因です。以下に、簡単な例で Verilog と SystemVerilog の比較をしてみます。

Verilog	SystemVerilog

`module bit_counter(data,bit_count);`
`input [7:0]  data;`
`output [3:0]  bit_count;`
`reg [3:0]  bit_count;`
 	
`always @(data)`
`  bit_count = count_ones(data);`
 	
`function [3:0] count_ones;`
`input [7:0]  data_in;`
`reg [7:0]  tmp;`
`begin`
`  count_ones = 0;`
`  tmp = data_in;`
`  while( tmp ) begin`
`  count_ones =`
`  count_ones + tmp[0];`
`  tmp = tmp >> 1;`
`  end`
`end`
`endfunction`
 	
`endmodule`
	`module bit_counter(`
	`input [7:0]  data,`
	`output logic [3:0]  bit_count);`
	
	`always_comb`
	`  bit_count = count_ones(data);`
	
	`function logic [3:0]`
	`  count_ones(logic [7:0] d);`
	`count_ones = 0;`
	`foreach(d[i])`
	`  if( d[i] )`
	`  count_ones++;`
	`endfunction`
	
	`endmodule`

この記述では、入力信号 data には 0 と 1 しか含まれていないと仮定しています。x や z が含まれていると、正しく動作しません。	入力信号 data に x や z が含まれていても正しく動作するように記述してあります。

18　無駄な命令　2024.08.17

Verilog でも SystemVerilog でも共通して言える事ですが、マンネリ化した考えで記述すると、無駄な命令を書いている可能性があります。特に定型化した処理法を見直すと効果があります。

具体例をもとにして論点を明確にします。以下は前回使用した Verilog 記述例です。

```
function [3:0] count_ones;
input [7:0]  data_in;
reg [7:0] tmp;
begin
    count_ones = 0;
    tmp = data_in;
    while( tmp ) begin
        count_ones = count_ones + tmp[0];
        tmp = tmp >> 1;
    end
end
endfunction
```

慣習化した記述法

変数 tmp のビットを右から左へ順に調べるために、シフトオペレータ（tmp >> 1）を使用していますが、本質的な記述法ではなく、以下のように他の命令でも実現できます。

```
tmp = tmp[7:1];
```

パートセレクトは、常に、符号なしなので、右辺は{1'b0,tmp[7:1]}と同じです。つまり、右辺は(tmp >> 1)を表現しています。

明示的に左辺と右辺のビット長を等しくするためには、以下のようにすれば良いです。

```
tmp = {1'b0,tmp[7:1]};
```

このように、古くから知られている記述法を見直すのも決して無駄ではないと言えます。

参考
Verilog (IEEE Std 1364-2005)と SystemVerilog (IEEE Std 1800-2023)においては、パートセレクトは、常に、符号なしです。

SystemVerilog のビットセレクト、パートセレクト、｛｝オペレータを巧みに組み合わせると、効率の良い演算を実装できます。

参考文献
SystemVerilog 超入門、共立出版 2023.

19　IEEE Std 1800-2023　2024.09.01

SystemVerilog は、進歩を続け IEEE Std 1800-2023 に至りました。その間、幾つかのリリースを経ていますが、重要な点を振り返ってみます。なお、今後の数年間は SystemVerilog の改訂はないと予想されるので、今が最新版の知識を習得する絶好の機会です。

先ず、重要な変遷は以下のようになります。

IEEE Std 1800-2005 → IEEE Std 1800-2009 → IEEE Std 1800-2012 →
IEEE Std 1800-2017 → IEEE Std 1800-2023

リリース	補足
IEEE Std 1800-2005	このリリースの SystemVerilog は IEEE Std 1364-2005 (Verilog HDL)の拡張言語であると宣言しているだけであり、SystemVerilog が Verilog HDL を基礎言語として包含しているとは明記していないようです。
IEEE Std 1800-2009	SystemVerilog は IEEE Std 1364-2005 (Verilog HDL)と IEEE Std 1800-2005 を統合した言語として定義されています。
IEEE Std 1800-2012	このリリースには SystemVerilog として機能が充実していますが、望ましくない機能も追加されています。例えば、現在では、非推奨となっているオペレータオーバーローディングの機能が含まれています。このリリースの LRM を読んだ方は、IEEE Std 1800-2017 以降のリリースの LRM を読む必要があります。
IEEE Std 1800-2017	非推奨機能等を除外して非常に良く書かれた LRM です。このリリースに書かれた知識を持てば、世界で通用する技術者と言えます。
IEEE Std 1800-2023	前仕様の解説上の改善と僅かですが新機能が含まれています。ただし、非常に難解な英文で書かれています。

余談

国内では、古くから SystemVerilog を導入している企業が多いと思いますが、技術者と話をして驚きました。SystemVerilog を使い始めた時期が古いせいか、2009 年以前の SystemVerilog 知識で業務に携わっている人が多いという印象を受けました。勿論、SystemVerilog の改変に合わせて知識を更新する人もいるでしょうが、しない人も多いと思います。SystemVerilog の変遷は、携帯電話がガラケーからスマフォに変化していく時代と重複するので、その表現を借りれば、いまだにガラケー的な知識で最先端のデザインを目指している人も多いと言えます。C/C++は一度学習すれば、その後はプログラミング技術の習得になりますが、SystemVerilog では仕様自体が進化を続けるので、常に学習が必要です。私の投稿を読んでいる方は最新の知識を持っているので、世界で通用します。自信を持って下さい。

20　implicit_data_type　2024.09.05

SystemVerilog は、Verilog シンタックスをサブセットとして包含する寛容な文法表現になっていますが、それを実現するために多くの工夫が組み込まれています。

工夫の一つとして、シンタックス上の変数 implicit_data_type があります。文法上の定義は以下のようになります。

```
data_declaration ::=
  [ const ] [ var ] [ lifetime ] data_type_or_implicit ...
data_type_or_implicit ::= data_type | implicit_data_type
function_data_type_or_implicit ::=
  data_type_or_void | implicit_data_type
implicit_data_type ::= [ signing ] { packed_dimension }
signing ::= signed | unsigned
```

要約すると、implicit_data_type は、データタイプが指定されずに signing または packed 次元が指定されれば、標準的なデータタイプ（logic）を仮定します。空の implicit_data_type の場合にも標準的なデータタイプが仮定されます。

Verilog には logic と呼ばれるデータタイプがないため、それを自動生成するために、implicit_data_type が効果的な役割をしています。以下の Verilog で書かれたファンクション定義を見ると、その働きを理解できます。

Verilog 記述と implicit_data_type

packed 次元を指定したファンクション	戻り値の型を指定しないファンクション
function [3:0] count_ones; input [7:0] data_in; reg [7:0] tmp; ... endfunction	function check; input [7:0] data_in; reg [7:0] tmp; ... endfunction
ファンクションの戻り値にデータタイプが指定されずに packed 次元だけが指定されているので、implicit_data_type のルールにより、戻り値は 4 ビットの logic になります。	ファンクションの戻り値に何も指定されていませんが、空の implicit_data_type のルールにより、戻り値は 1 ビットの logic 型となります。
data_in にはデータタイプが指定されずに packed 次元だけが指定されているので、implicit_data_type のルールにより、data_in は 8 ビットの logic 型となります。	

参考
変数の定義時には、空の implicit_data_type を使用できません。以下のようにキーワード var を使用します。ネットの場合には、空の implicit_data_type を使用できます。

```
var    v;  // OK
       a;  // ILLEGAL
wire   w;  // OK
```

21　アレイ計算メソッド　2024.09.13

SystemVerilog のアレイ計算メソッド（array reduction methods）には使用制限があるので注意して下さい。

使用制限は以下のようになります。アレイの sum()、product()、and()、or()、xor()等のメソッドは、整数系のアレイにしか適用できません。

LRM 176 ページ
Array reduction methods may be applied to any unpacked array of integral values to reduce the array to a single value.

参考

- アレイ計算メソッドはアレイ要素のデータタイプの値を戻すので、演算が定義されているデータタイプに限定されます。実数には加算と乗算が定義されていますが、AND、OR、XOR が定義されていないので、実数のアレイに計算メソッドを適用できません。つまり、整数系のアレイにのみアレイ計算メソッドを適用できます。
- 検索系のメソッド（find、find_*等）は結果をキューとして戻すので、アレイ要素のデータタイプに依存せずに全ての unpacked アレイに使用できます。

□

以下のように使用できます。

```
byte   b[] = { 1, 2, 3, 4 };
int    y;
y = b.sum ;                    // y becomes 10 => 1 + 2 + 3 + 4
y = b.product ;                // y becomes 24 => 1 * 2 * 3 * 4
y = b.xor with ( item + 4 );   // y becomes 12 => 5 ^ 6 ^ 7 ^ 8
```

アレイが整数系以外であれば、計算メソッドを使用できないため、foreach を使用します。

```
typedef struct { real field1; bit field2; } T;

function automatic T Tsum (input T driver[]);
   Tsum.field1 = 0.0;
   foreach (driver[i])
      Tsum.field1 += driver[i].field1;
endfunction
```

22　文　2024.09.17

SystemVerilog の文はどのような語または記号で始まるか明確に定義できますか？意外と面白いクイズです。

通常使用する文の先頭に来る語の概略を述べると以下のようになりますが、キーワードだけではない事がすぐわかります。

- キーワード
- ユーザが定義したデータタイプ
- インスタンス用のモジュー名（例： `adder DUT(.*);`）
- インスタンス用のプログラム名
- インスタンス用のインターフェース名
- インスタンス用のチェッカー名
- パッケージ名（例： `pkg::print();`）
- クラス名（例： `X::count++;`）
- クラスハンドル名
- クロッキングブロック名
- カバーグループ名
- カバーグループのインスタンス名
- ユーザが定義したタスクおよびファンクション名（呼び出し）
- システムタスクとファンクション名（呼び出し）
- 変数名、ネット名
- 文のラベル名（例： `p1: a = 0;`）
- カバーポイント名（例： `cp: coverpoint x;`）
- クロスカバレッジ名（例：`aXb : cross a, b {...}`）
- `{`、`;`、`@`、`#`、`->`
- 等々（この他にもあれば付け足して下さい）。

例えば、`{` 、`;` 、`->` は、以下のように文の先頭で使用されます。

```
{co,sum} = a + b;
->ev;
;
```

23　IEEE Std 1800-2023　2024.10.18

IEEE Std 1800-2023 で追加された機能を積極的に使用すると、記述がより汎用的になる場合があります。

type(this)を利用して UVM クラスをより汎用的に記述する例を紹介します。モニターでは TLM ポートを定義しますが、クラス名を指定しなければならない場合があります。クラス名を省略できればより汎用的になるので、type(this)を活用できます。

```
class simple_monitor_t extends uvm_monitor;
uvm_analysis_imp#(simple_item_t,simple_monitor_t) receive_port;
uvm_analysis_port #(simple_item_t)                send_port;
...
endclass
```

type(this)を使用すると以下のようになります。この方が、依存性が少ないので望ましいのは明かです。

```
class simple_monitor_t extends uvm_monitor;
uvm_analysis_imp#(simple_item_t,type(this)) receive_port;
uvm_analysis_port #(simple_item_t)          send_port;
...
endclass
```

参考
マクロの引用やマクロ定義内に type(this)を使用する場合には注意をした方が良いです。

24　戻り値　2024.10.31

IEEE Std 1800-2023 では、条件判定結果が int から 1 ビットの logic に変わりました。例えば、SystemVerilog では、a < b の結果は 1'b1、1'b0、1'bx の何れかです。しかし、戻り値が真または偽になるメソッドでも、依然として int を戻す場合があるので、注意が必要です。

associative アレイには、first()、last()、next()、prev()等のメソッドが準備されています。これらのメソッドは、該当するキーが存在すれば 1、存在しなければ 0 を戻すので、戻り値を条件判定結果のように 1 ビットの logic で表現できるように思えるかも知れませんが、残念ながら int のままでなければなりません。何故なら、以下の理由によります。

LRM 168 ページ
The argument that is passed to any of the four associative array traversal methods first(), last(), next(), and prev() shall be assignment compatible with the index type of the array. If the argument has an integral type that is smaller than the size of the corresponding array index type, then the function returns –1 and shall truncate in order to fit into the argument.

これらのメソッドを使用すると引数には該当するキーが設定されますが、キーの型が整数系の場合には、精度が十分保証されない場合があります。例えば、以下の場合には first(ix) における ix は精度が不十分です。したがって、その状態を表現するために status は-1 を受けとります。しかし、1 ビットの logic では-1 を表現できません。したがって、これらのメソッドの戻り値を表現するためには、int でなければなりません。

```
string     aa[int];
byte    ix;
int     status;
aa[1000] = "a";
status = aa.first(ix);
```

この注意点は、下記の文献にもあるので、思い出して下さい。

参考文献
SystemVerilog 超入門、共立出版 2023、第 3.6.1 項

25　基礎の確認　2024.11.14

設計分野では SystemVerilog の一部の機能しか使用されませんが、それでも理解しておかなければならない事が多いです。

初心者は以下のチェックリストを基にして理解度を確認すると良いです。その他にも追加して下さい。良い結果であれば、実務に励めます。結果が思わしくなければ『SystemVerilog 超入門』を復習すれば良いです。

(1) ネットと変数の相違は何ですか？
(2) logic と reg の相違はありますか？
(3) int と integer の相違は何ですか？
(4) integer と time の違いは何ですか？
(5) int と bit signed [0:31]との相違はありますか？
(6) logic signed [7:0] a;において a と a[7:0]は同じですか？
(7) @a と@(posedge a)の相違はありますか？
(8) always_comb と always @(*)の相違は何ですか？
(9) always_comb 内でタスクを呼び出すとどうなりますか？
(10) #10 a = b;と a = #10 b;の相違は何ですか？
(11) a = b;と#0 a = b;の相違は何ですか？
(12) #0 a = 1;と a <= 2;では、どちらが先に a に値を設定しますか？
(13) module では、initial と always のどちらが先に実行を開始しますか？
(14) module の initial と program の initial はどちらが先に実行を開始しますか？
(15) module の最初のポートの方向が省略されると、方向はどうなりますか？
(16) タスクやファンクションで最初のポートの方向が省略されると、方向はどうなりますか？
(17) module の inout ポートはネットですか変数ですか？
(18) module の ref ポートはネットですか変数ですか？
(19) ファンクションの定義で戻り値の型を省略すると戻り値の型はどうなりますか？
(20) module 内のタスクやファンクションで変数を宣言すると、その変数は常に存在しますか、それとも呼ばれる度に変数の領域が確保されますか？
(21) wire、uwire、wand、wor 等のネットと trireg との大きな違いは何ですか？
@(posedge clk)において、clk が 1'b0 から 1'bx に変化すると、posedge イベントは起きますか？

26　ショートサーキットの活用　2024.11.23

SystemVerilog のオペレータ&&と||がショートサーキットである事は良く知られています。この機能は記述の最適化に効果があります。

少し人為的ですが、わかり易い例を使用してショートサーキットの威力を紹介します。変数 a と b が以下のように定義されているとします。

```
logic  a, b;
```

次に、以下のような if 文があると仮定します。実は、この記述は最適化されていません。

```
if( a == 1'b1 || (a == 1'b0) && (b == 1'b1) )
```

最適化すると以下のようになります。

```
if( a|b )
```

何故なら、||はショートサーキットなので、a == 1'b1 が真であれば、以下の条件は実行されません。また、実行される時には、a == 1'b0 が真なので、最適化されます。

```
(a == 1'b0) && (b == 1'b1)
```

つまり、これは以下のように最適化されます。

```
b == 1'b1
```

したがって、与えられた記述は以下のように簡略化されます。

```
if( a == 1'b1 || b == 1'b1 )
```

これは、以下の記述に等しくなります。

```
if( a|b )
```

このように、条件が&&や||で結ばれている時には、見直すと効果が出てくる場合があります。勿論、見直さなくても論理合成が最適化をしてくれるので心配はありません。

参考文献

SystemVerilog による効果的実装技術、アートグラフィックス 2024.

27　reg　2024.11.30

SystemVerilog では、reg を使用しない習慣が推奨されていますが、実際問題として、logic と reg が全く同じように使用できるわけではありません。

先ず、SystemVerilog のルールを引用しておきます。

LRM 103 ページ
A net type keyword shall not be followed directly by the reg keyword.

ネットタイプと logic を一緒に指定できますが、ネットタイプに reg が続くとエラーになります。

```
tri reg            r;     // error
wire reg [15:0]    v;     // error
```

しかし、以下のような宣言は可能です。

```
var reg            r;     // OK
```

更に、状況を複雑にさせるのは以下のルールです。

LRM 103 ページ
The reg keyword can be used in a net or port declaration if there are lexical elements between the net type keyword and the reg keyword.

ネットタイプと reg の間にシンタックス要素が存在すれば、エラーにはなりません。

```
wire scalared reg    r;     // OK
```

28　アレイメソッド　2024.12.06

SystemVerilog のアレイメソッドは予想以上に強力です。

簡単な例を紹介します。

```
int    a[];
```

アレイ a には負でない整数が記録されているとして、奇数が 1 から順に並んでいるかを以下のようにして調べられます。

```
if( a.sum(x,i) with (int'(x == 2*x.i+1)) == a.size )
    $display("a[] is correct");
else
    $display("a[] is incorrect");
```

map を使用できますが、少し冗長になります。別のアレイを作る必要はないので、上記の方法の方が良いと言えます。ただし、map 方式によれば、a のどの要素が奇数ではないかが分かるので便利と言えます。

```
int    b[];
b = a.map(x,i) with( x == 2*x.i+1 ? 1 : 0);
if( b.sum() == a.size() )
...
```

しかし、この場合には、以下のように find_index()を使用するのが得策と思えます。q には、奇数のルールに違反する要素のインデックスが戻されるので、どの要素に問題があるかが分かります。map バージョンよりも効率が良いと考えられます。

```
int    q[$];
q = a.find_index(x,i) with(x != 2*x.i+1);
if( q.size() == 0 )
...
```

この他の方法も試して下さい。

29　アレイメソッド　2024.12.12

SystemVerilog のアレイ操作メソッドで使用できるインデックスイタレータは、一般的には int 型ですが、associative アレイの場合にはキーの型に合わされます。

例えば、以下の場合の index は int 型です。x の型は enum の color_e です。

```
typedef enum { RED, GREEN, BLUE, WHITE, BLACK, YELLOW }  color_e;
color_e   a[] = '{ RED, BLUE, GREEN, YELLOW, WHITE },
          q[$];
...
q = a.find(x) with(x.index inside {[1:2]});
```

以下の場合には index はクラスオブジェクトを指しています。item の型は int です。

```
class sample_t;
string name;
function new(string name);
   this.name = name;
endfunction
endclass

int    a[sample_t], q[$];
...
q = a.find() with (item.index.name == "driver");
```

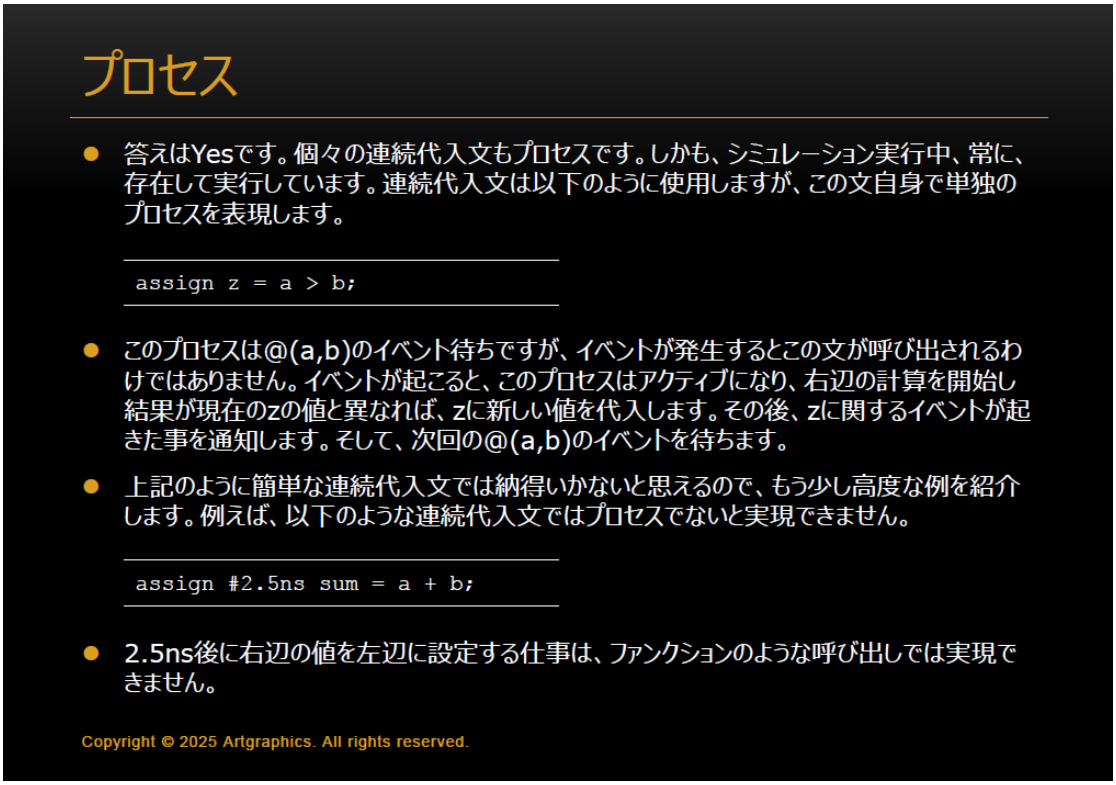
実際問題として、かなり高度な記述が可能です。

30　SystemVerilog 規格　2024.12.21

まだ知らない人のために、SystemVerilog 規格の変遷を分かり易く纏めました。

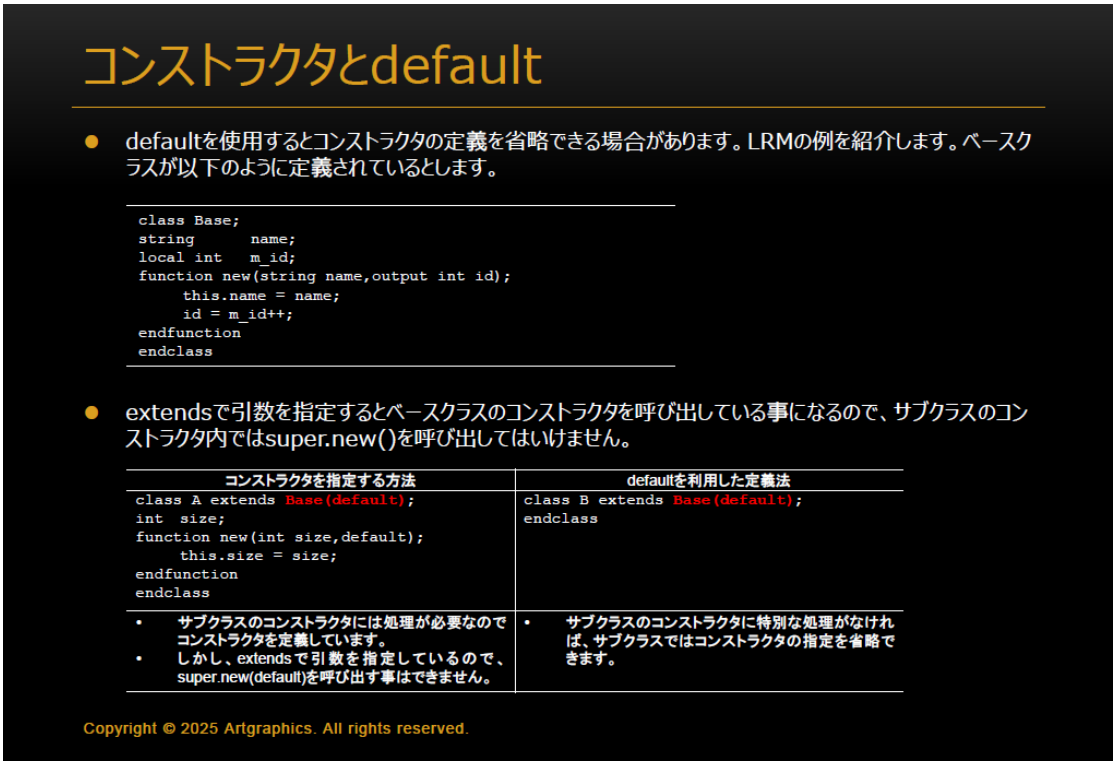
31　プロセス　2025.01.17

SystemVerilog の `initial` と `always` 系がプロセスである事は承知していると思います。その他、`fork` で生成されるプロセスがあるのも知っていると思います。この他にもプロセスがあると思いますか？



32　default　2025.01.31

SystemVerilog では、ベースクラスを `extends` する際に引数を指定するとコストラクタの定義は簡略化されます。特に、`default` を指定するとサブクラスではコンストラクタの定義を省略できます。



33　コンストラクタと default　2025.02.01

SystemVerilog では extends する際に引数を指定できますが、指定すると暗黙にベースクラスのコンストラクタを呼び出している事になります。

従って、この場合にはサブクラスのコンストラクタ内では super.new()を呼び出す事はできません。以下の場合には、エラーになります。

```
class B extends Base(default);
int    size;
function new(int size,default);
   super.new(default);     // ILLEGAL
   this.size = size;
endfunction
endclass
```

そもそも、Base(default)でベースクラスのコンストラクタを呼び出しているので、コンパイラーにはすべての知識が与えられています。つまり、コンパイラーは、コンストラクタの先頭に super.new(default)を先頭に挿入する知識を持っています。このため、ユーザがわざわざ super.new(default)を指定する必要はありません。

参考
super.new(...)を指定しても良いとなっていると、コンパイラーの仕事はかなり複雑になります。コンパイラーを開発している人にとっては嬉しい制限事項です。

34　カバーグループ　2025.03.22

SystemVerilog のサブクラスでカバーグループを拡張できますが、クラスを拡張する場合と多少異なります。サブクラスではカバーグループのインスタンスを作らずに、ベースクラスのコンストラクタを呼び出して作って貰います。

以下の例では、sub_t クラスで base_t のカバーグループ cg を拡張しています。sub_t の cg のインスタンスを作るために、ベースクラスのコンストラクタを呼び出しています。

```
class base_t;
rand bit [2:0]   a;
covergroup cg(int low,int high);
    coverpoint a { ignore_bins value ={ [low:high] }; }
endgroup
function new(int low,int high);
    cg = new(low,high);
endfunction
endclass
```

```
class sub_t extends base_t;
rand logic [3:0] value;
covergroup extends cg;
    coverpoint value { bins value_bins[4] = {[0:15]}; }
endgroup
function new(default);
    super.new(default);
endfunction
endclass
```

ベースクラスのコンストラクタが cg のインスタンスを作ると、それは sub_t 用のインスタンスを作ることになります

言い換えれば、以下に示す new が恰も virtual コンストラクタのように動作します。

```
cg = new(low,high);
```

35　inside オペレータ　2025.03.29

case/casex 文を使用すると比較式を一度指定するだけで済みますが、if 文や式では比較式を何度も指定しがちです。特に、比較が OR(||)を伴う時には冗長な表現になります。そのような時には、SystemVerilog では inside オペレータを使用すると効果的です。

一般的に言えば、case/casex 文を使用すると OR 条件を簡潔に表現できます。一方、if 文または式では、同じ条件式を羅列する傾向にあります。例えば、以下のように{a,b,c}を何度も指定しがちです。

```
z = {a,b,c} === 3'b110 || {a,b,c} == 3'b101 || {a,b,c} == 3'b011;
```

SystemVerilog には便利な inside オペレータがあります。

case/casex 文	inside オペレータ
case ({**a,b,c**}) 3'b110, 3'b101, 3'b011: z = 1'b1; default z = 1'b0; endcase	z = {a,b,c} **inside** {3'b110,3'b101,3'b011};
casex ({**a,b,c**}) 3'b11x, 3'b1x1, 3'bx11: z = 1'b1; default z = 1'b0; endcase	z = {a,b,c} **inside** {3'b11?,3'b1?1,3'b?11};

参考文献

SystemVerilog による設計と論理合成、アートグラフィックス 2025

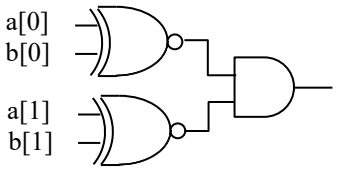
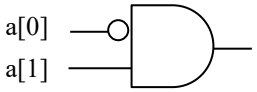
36　リテラル　2025.04.03

ハードウェア記述言語に使用されるリテラルは、ハードウェア種別を限定する機能を持ちます。SystemVerilog により例示します。

要点は以下のようになります。

シーケンシャル回路の場合には、リテラルはフリップフロップ種別を指定する機能を持ち、組み合わせ回路の場合にはプリミティブな回路に限定する機能を持ちます。

組み合わせ回路の場合には、より具体的になるので最適化されます。

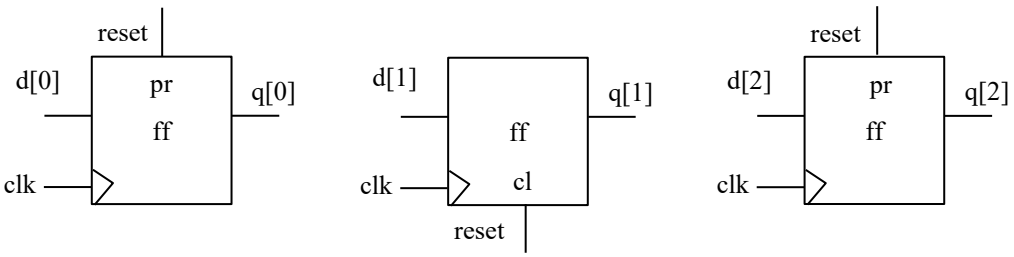
a == b	a == 2'b10
	

逆に、シーケンシャル回路の場合にはリテラルを使用すると複雑になります。以下の記述は、微妙に異なるフリップフロップを生成します。

```
module pdff(input clk,reset,[2:0] d,output logic [2:0] q);

always_ff @(posedge clk,posedge reset)
    if( reset )
        q <= 5;
    else
        q <= d;

endmodule
```



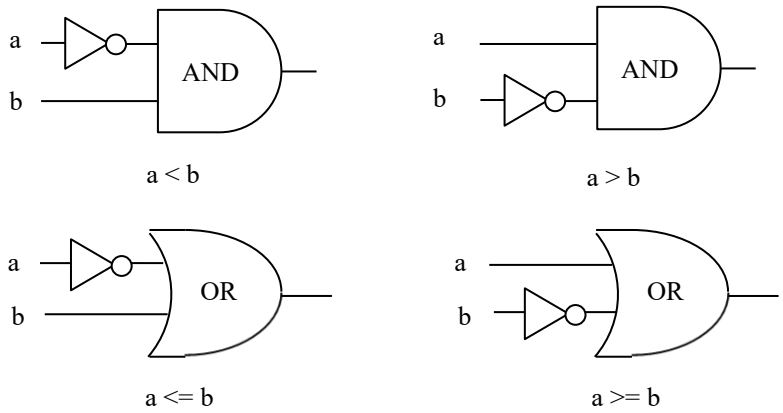
参考文献

SystemVerilog による設計と論理合成、アートグラフィックス 2025

37　複合回路　2025.04.05

AND、OR、INV は素朴な機能を持つ回路ですが、AND と OR に INV を活用すると複合機能を持つ回路に変化します。SystemVerilog を使用して、簡単な例を示します。

先ず、AND と OR に INV を付け加えると複合機能の回路に変化します。



例えば、a <= b を例にとれば、「a が真であればb は真である」を意味しています。しかも、a が偽である場合には、常に、成立するので b はどうでも良い事になります。つまり、SystemVerilog の以下の式を意味します。この演算子は、logical implication として知られています。

$$a -> b$$

このように、簡単な論理回路でも論理を表現するツールとして有効に活用できます。

参考文献
SystemVerilog による設計と論理合成、アートグラフィックス 2025

38　サイクルディレー　2025.04.10

SystemVerilog による検証となるとピンからキリまでの機能を使えます。ピンといえば、アサーションやファンクショナルカバレッジかも知れません。キリといえば、位が低い意味ではなく細かな制御をできる意味において、シーケンスとディレーがあります。特に、サイクルディレーは便利です。

複数の信号値が変化するイベントを順に待つ場合には、以下のようにサイクルディレーを使用すると明瞭になります。

```
logic  clk = 0,
       a, b, c;

sequence check_abc;
   @(posedge clk) $rose(a) ##2 $fell(b) ##1 $rose(c);
endsequence

initial begin
   @check_abc $display("@%3t: saw a-b-c",$time);
   ...
end
```

@check_abc では a、b、c の順に信号が変化するイベントをとらえています。このように、複数のクロッキングイベントの状態を厳密に定義するためにシーケンスとサイクルディレーを使用できます。

参考文献

SystemVerilog による検証の基礎、森北出版 2020.

39　DUT の検証　2025.04.11

組み合わせ回路であろうとシーケンシャル回路であろうと、シミュレータの観点からは、always @(...)で書かれた回路です。その回路の検証は、always @(...)の待ち状態が有効になってからでないと開始できません。その原理原則を保証する機能は SystemVerilog の fork/join_none です。

どんなに複雑なシステムでも、fork/join_none により DUT が待ち状態に入ってから検証環境が実行を開始するようにできます。UVM がその最たる例です。

どのような記述でも同じなので、下記のような簡単な検証環境で解説します。driver と collector のプロセスが生成されていますが、それらのプロセスは直ぐには実行を開始しません。

```
initial begin
    fork
        driver();
        collector();
    join_none
end
```

initial や always 等の全てのプロシージャが終了するか待ち状態に入って、且つ、活動するプロセスが存在しなくなった時点で初めて、driver と collector プロセスに実行許可が出ます。つまり、これらの検証プロセスが実行を開始する時には、DUT はイベントを待ち状態にあります。したがって、driver は DUT をドライブする事ができます。

一方、collector は DUT の出力をモニターするので、driver が DUT をドライブする時には collector が待ち状態に入っていなければならないのですが、driver は collector とは独立しているので collector の準備が完了している保証はありません。それにも関わらず、collector は DUT からの出力を正しくサンプリングできます。何故なら、collector はクロッキングブロック（cb）を利用するからです。collector は@cb のタイミングで DUT の出力をサンプリングします。イベント待ち@cb は$time==0 の Active 領域で有効になります。その後、DUT のイベント待ちが解除されて DUT の処理が再開します。イベント待ち@cb は Observed 領域で解除されるので、その時には、ドライバーおよび DUT の活動は完了しています。すなわち、collector は DUT からの出力を安全に取り出すことができます。

参考文献

SystemVerilog シミュレーションの論理、アートグラフィックス 2024.

40　アレイ　2025.04.26

SystemVerilog の初歩的な機能は意外と忘れがちです。この際、思い出しておきましょう。「連休は家にいる」という人が 37%だそうです。もし 37%派であれば復習してください。

例題―1　以下のアレイに対して、全ての要素に"red"を設定してください。

```
string        color[1024];
```

解答

```
color = '{default:"red"};
```

foreach 等を使用しないようにしましょう。
■

例題―2　以下の packed アレイに対して、全ての要素に 1 を設定してください。

```
logic [1023:0]   a;
```

解答

```
a = '1;
```

この記法はスケーラブルです。つまり、アレイのサイズに依存しません。
■

例題―3　以下の packed アレイに対して、MSB と LSB に 1、それ以外には 0 を設定してください。

```
logic [1023:0]   a;
```

解答

```
a = '{ 1023:1, 0:1, default:0 };
```
■

41　ハーフアダー　2025.05.15

【初心者向け】簡単なクイズ（むしろ、なぞなぞ）を試してみましょう。1 ビットの信号 a と b があるとします。それらの信号の AND、XOR、および和を求める計算を SystemVerilog の一行で表現してください。

解答

AND と XOR を表す変数をそれぞれ z_and、z_xor とします。ハーフアダーは AND と XOR で構成されているので、それを活用すれば良い事になります。すると、以下のように計算できます。

```
assign {z_and,z_xor} = a + b;
```

z_and は AND、z_xor は XOR、{z_and,z_xor}は a と b の和になっているので問題は解けました。あるいは、以下のようにもできます。

```
assign {z_and,z_xor} = {a&b,a^b};
```

参考文献

SystemVerilog による設計と論理合成、アートグラフィックス 2025.

42　ビット演算　2025.06.06

ビット処理をループで記述するのも珍しい事ではありませんが、気を付けないと冗長な表現をする事があります。SystemVerilog の例を紹介します。

以下のような状況を仮定します。

```
logic [7:0]  a, b, z;
```

最近では、次のような記述をする事はめったにありません。

```
for( int i = 0; i < 8; i++ )
          z[i] = a[i] & b[i];
```

これは、以下のように 1 行で書けるからです。

```
z = a & b;
```

では、以下のような場合はどうでしょうか？異なるインデックス式が必要なので for ループを使用して簡単に済ませています。

```
z[7] = a[7] ^ b[7];
for( int i = 6; i >= 0; i-- ) begin
   if( i < 4 )
      z[i] = a[i+1] & b[i];
   else
      z[i] = a[i+1] | b[i];
end
```

もっともらしい for ループの仕方なので特別な異論はないと思いますが、全てがビット処理なので、以下のようにループをせずに記述できます。

```
z = {a[7]^b[7],a[7:5]|b[6:4],a[4:1]&b[3:0]};
```

43　"""　2025.06.13

何行にも渡るメッセージを文字列として定義するためには、SystemVerilog では"""..."""を使用すると良いです。

ログファイルの一部からの文字列を引用したりする場合には、何行にもメッセージが渡るので改行コードを挿入するのが厄介です。そのような場合には"""..."""を使用すると便利です。例えば、以下のようなメッセージを文字列に定義する場合を考えてみます。

```
w=abcd m_byte=cd m_word=abcd
w=ab56 m_byte=56 m_word=ab56
w=0123 m_byte=23 m_word=0123
```

この場合には、メッセージをクリップボードにコピーして、文字列の変数に以下のように貼り付ければ良いです。メッセージ間の改行コードは自動的に処理されます。

```
string s, test_result =
"""w=abcd m_byte=cd m_word=abcd
w=ab56 m_byte=56 m_word=ab56
w=0123 m_byte=23 m_word=0123""";
```

こうすると、以下のように test_result を通常の文字列として扱えます。

```
if( s == test_result )
...
```

44　ショートサーキット　2025.06.21

SystemVerilog にはショートサーキット評価をするオペレータがある事をご存じだと思います。そのショートサーキットオペレータはシミュレータだけのものではなく論理を最適化する手段でもあります。

以下に示す簡単な記述例を見て下さい。&&オペレータはショートサーキットなので、記述には無駄がある事が分かります。

```
module design(input a,b,output z);
assign z = a && a == b;
endmodule
```

ショートサーキット評価の定義により&&オペレータの左オペランドが偽であれば右オペランドを評価しません。つまり、左オペランドが真の時のみ右オペランドを評価します。この時には a==1'b1 なので、a==b は過剰な条件式です。すなわち、b==1'b1 で十分です。したがって、以下のように書き換えられます。

```
assign z = a && b;
```

このように、ショートサーキットを活用すると記述が簡潔になります。因みに、||オペレータと&&オペレータ以外にもショートサーキット効果を持つ記述があります。これは、条件付きショートサーキットとも呼べる記述法です。

```
module comparator #(NBITS=4)
     (input [NBITS-1:0] a,b,output logic gt,lt,eq);
always @(a,b) begin
    {gt,lt,eq} = 3'b001;
    for( int i = NBITS-1; i >= 0; i-- ) begin
        if( a[i] != b[i] ) begin
            {gt,lt,eq} = {a[i],~a[i],1'b0};
            break;
        end
    end
end
endmodule
```

条件付きショートサーキット

ショートサーキットの解説は文献[1]の 4.1.8 項にあります。ショートサーキットの効果により上記の comparator の for 文がどのように展開されるかは文献[2]に詳しい解説があります。

参考文献

[1] SystemVerilog 超入門、共立出版 2023.
[2] SystemVerilog による設計と論理合成、アートグラフィックス 2025.

45　フルアダー　2025.06.27

条件判定の一部としてアダーを使用すると、アダーの一部の機能しか活用されない場合があります。SystemVerilog で例を紹介します。

例えば、以下のような例を考察します。そもそも、アダーを使用する必要はありません。

```
module design(input a,b,c,output z);
assign z = a+b+c >= 2;
endmodule
```

a+b+c はフルアダーですが、実は、フルアダーの一部の機能しか活用されていません。したがって、無駄な回路が生成されている記述と考えられます。以下に理由を解説します。

与えられた式を以下のように書き換えると分かり易くなります。

```
assign z = a+b+c inside {2,3};
```

a+b+c はフルアダーなので、{co,sum}と置き換えられます。すると、右のような真理値表を得ます。この真理値表から、与えられた式は以下のようになります。

co	sum	z
0	0	0
0	1	0
1	0	1
1	1	1

```
assign z = co;
```

要約すると、フルアダーの co だけが利用され、sum は無駄になっています。したがって、フルアダーを使用せずに以下のように表現すべきであったと言えます。

```
module design(input a,b,c,output z);
assign z = a&b | b&c | c&a;
endmodule
```

以上から、アダーを条件式の一部として使用している場合には、記述を見直す必要があります。co は majority 関数と呼ばれ重要な組み合わせ回路を表現します。

参考文献

[1] SystemVerilog 超入門、共立出版 2023.
[2] SystemVerilog による設計と論理合成、アートグラフィックス 2025.

46　参考文献

かつての Verilog HDL の時代では、中途半端な HDL の知識でもデジタルシステムの設計や検証を何とかできましたが、近年ではチップの高集積度と高性能化の傾向と相まって設計および検証段階で厳密な追及が不可欠になりました。正に、『生兵法は大怪我のもと』とならないように、SystemVerilog に関する正しい知識を習得する必要があります。文献[3-7]は純正の SystemVerilog 参考書です。SystemVerilog が備えている機能を正しく使用するために有益な書物です。

設計および検証作業で必要となる SystemVerilog の基礎知識に関しては、『SystemVerilog 超入門』をお読みください。この書物は SystemVerilog の基本機能を非常に詳しく解説しているので、初心者におすすめします。

[1] IEEE Std 1800-2017: IEEE Standard for SystemVerilog – Unified Hardware Design, Specification and Verification Language.
[2] IEEE Std 1800-2023: IEEE Standard for SystemVerilog – Unified Hardware Design, Specification and Verification Language.
[3] 篠塚一也、SystemVerilog 超入門、共立出版 2023.
[4] 篠塚一也、SystemVerilog 入門、共立出版 2020.
[5] 篠塚一也、SystemVerilog による検証の基礎、森北出版 2020.
[6] 篠塚一也、実践 UVM 入門、森北出版 2021.
[7] 篠塚一也、検証のための SystemVerilog プログラミング、森北出版 2022.
[8] SystemVerilog による設計と論理合成、アートグラフィックス 2025.
[9] SystemVerilog IEEE Std 1800-2023 の概要、アートグラフィックス 2024.