

SystemVerilog 雑談

Document Revision: 9.5, 2026.05.11

アートグラフィックス

篠塚一也



SystemVerilog 雑談

© 2026 アートグラフィックス
〒124-0012 東京都葛飾区立石 8-14-1
www.artgraphics.co.jp

SystemVerilog Monologue

© 2026 Artgraphics. All rights reserved.
8-14-1, Tateishi, Katsushika-ku, Tokyo, 124-0012 Japan
www.artgraphics.co.jp

注意事項

- 弊社の技術資料の内容の一部、あるいは全部を無断で複写、複製、転載する事は、禁じます。
- 弊社の技術資料の譲渡、転売、模倣、又は、改造等の行為を禁止致します。

はじめに



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分からない情報、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

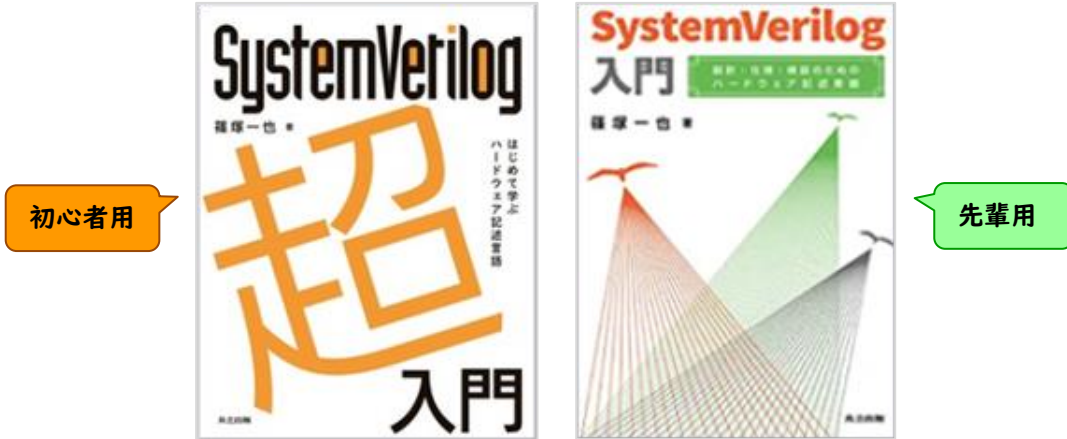
1	SystemVerilog 超入門	2023.06.27	1
2	IEEE Std 1800-2023	2024.08.30	2
3	非推奨機能	2022.04.18	3
4	ファンクショナルカバレッジ	2022.09.16	4
5	カバーポイント	2022.11.11	5
6	posedge と negedge	2023.06.09	6
7	ショートサーキット評価	2023.07.01	7
8	Inside	2024.01.19	8
9	キュー	2024.05.09	9
10	制約	2024.06.04	10
11	規定幅を持つ整数系	2024.06.25	11
12	SystemVerilog シミュレーション原理	2024.07.01	12
13	連鎖	2024.07.03	13
14	String	2024.07.07	13
15	default	2024.07.10	13
16	クラス	2024.08.14	14
17	Verilog	2024.08.15	15
18	無駄な命令	2024.08.17	16
19	IEEE Std 1800-2023	2024.09.01	17
20	implicit_data_type	2024.09.05	18
21	アレイ計算メソッド	2024.09.13	19
22	文	2024.09.17	20
23	IEEE Std 1800-2023	2024.10.18	21
24	戻り値	2024.10.31	22
25	基礎の確認	2024.11.14	23
26	ショートサーキットの活用	2024.11.23	24
27	reg	2024.11.30	25
28	アレイメソッド	2024.12.06	26
29	アレイメソッド	2024.12.12	27
30	SystemVerilog 規格	2024.12.21	28
31	プロセス	2025.01.17	29
32	default	2025.01.31	30
33	コンストラクタと default	2025.02.01	31
34	カバーグループ	2025.03.22	32
35	inside オペレータ	2025.03.29	33
36	リテラル	2025.04.03	34

37	複合回路 2025.04.05	35
38	サイクルディレイ 2025.04.10	36
39	DUT の検証 2025.04.11	37
40	アレイ 2025.04.26	38
41	ハーフアダー 2025.05.15	39
42	ビット演算 2025.06.06	40
43	””” 2025.06.13	41
44	ショートサーキット 2025.06.21	42
45	フルアダー 2025.06.27	43
46	保守・拡張性 2025.08.08	44
47	簡略記法 2025.08.24	45
48	for 文 2025.09.13	46
49	case/casex 2025.09.20	47
50	知識と技術 2025.09.25	48
51	キュー 2025.10.04	49
52	SystemVerilog 記述 2025.10.23	50
53	サブクラス 2025.10.27	51
54	乱数発生 2025.10.29	52
55	真理値表と case 文 2025.11.03	53
56	case+inside≠casex 2025.11.05	54
57	case+inside≠casex 2025.11.07	55
58	サブクラスでの制約定義 2025.11.13	56
59	SystemVerilog 入門の中国語版 2025.11.15	57
60	SystemVerilog 2025.12.04	58
61	SystemVerilog 2025.12.29	59
62	Gray コード 2025.12.31	60
63	実数型のインクリメントとデクリメント 2026.01.12	61
64	オペレータの結合法則 2026.01.12	62
65	inside 2026.01.20	63
66	attribute_instance 2026.02.02	64
67	計算オペレータ 2026.02.07	65
68	ダイナミックアレイ 2026.02.13	66
69	引数としてのファンクション 2026.02.25	67
70	積分 2026.02.26	68
71	FSM 2026.03.12	69
72	HDL 2026.03.28	70
73	final class 2026.04.02	71

74	final method 2026.04.03.....	72
75	実数型のランダム変数 2026.04.10	73
76	ファンクショナルカバレッジ 2026.04.12.....	74
77	カバーグループ 2026.04.17.....	75
78	type(expr) 2026.04.22.....	76
79	RNG 2026.04.24.....	77
80	RNG 2026.04.25.....	78
81	String 2026.05.05.....	79
82	参考文献.....	80

1 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と、読み易い体裁になっています。



2 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>



3 非推奨機能 2022.04.18

SystemVerilog には、使用しない方がよい機能もあります。

SystemVerilog 豆知識

機能	仕様と注意点
defparam	<ul style="list-style-type: none"> 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ため、使用しない事を勧めます。
procedural assign/deassign	<ul style="list-style-type: none"> 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ため、使用しない事を勧めます。
operator overloading	<ul style="list-style-type: none"> 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。

4 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.

5 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] の二つのピンしか生成されません。本来、(a+b) は 0~2 の値を取るなので三つのピンが必要なため、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。

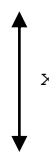
```
covergroup cg;
  a_plus_b: coverpoint (a+b+2'b0) iff (coverage_enabled)
  { bins fc_a_plus_b[] = {[0:2]}; }
endgroup
```

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.

6 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  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。

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
<pre>result = regA & (regB regC);</pre>	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
<pre>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</pre>	<pre>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</pre>
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。

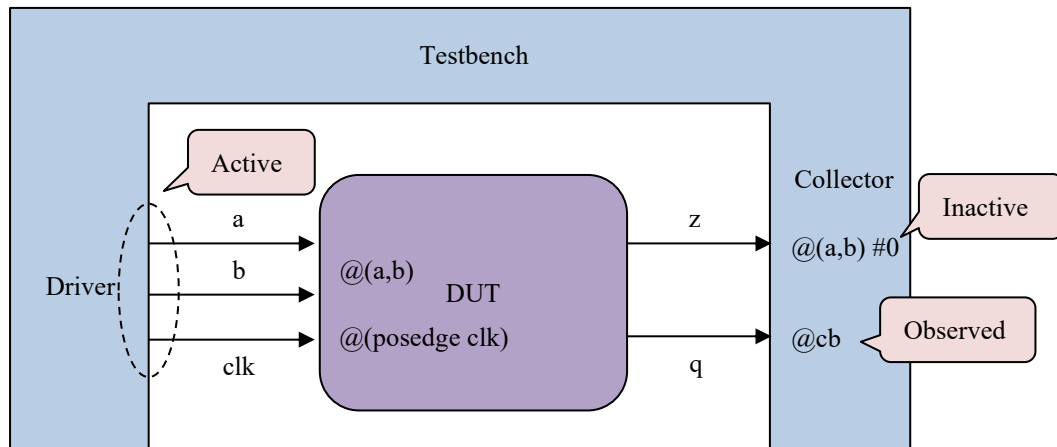
標準的な宣言	マッチする表現
int value;	bit signed [31:0] value;

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、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
<pre>string words[int] = '{default:"hello"}; logic [3:0] a[5]; a = '{default:2};</pre>	標準値の指定をする。
<pre>casex({sa,sb,sc}) 3'b1xx: out = a; 3'b01x: out = b; 3'b001: out = c; default out = d; endcase</pre>	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。
<pre>class sub1_t extends base_t; byte m_value; function new(default,byte v); super.new(default); m_value = v; endfunction endclass</pre>	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。
<pre>x dist {[100:102]:/3,default:/1};</pre>	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/casex 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	こうしておくで、UVM が改訂されて <code>build_phase()</code> の仕様に変更があるとコンパイラが自動的にエラーを発行してくれます。もし :extends が無いと、 <code>build_phase()</code> の仕様が異なってもエラーが出ません。しかも、仕様異なる場合、このクラスに定義されている <code>build_phase()</code> 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	<code>get_and_drive()</code> は UVM に定義されていない事を仮定しているのので、 :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
<pre> 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 </pre>	<pre> 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 </pre>
<p>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。</p>	<p>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。</p>

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っていますが、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。

まず、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ません。
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ホ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
<pre> function [3:0] count_ones; input [7:0] data_in; reg [7:0] tmp; ... endfunction </pre>	<pre> function check; input [7:0] data_in; reg [7:0] tmp; ... endfunction </pre>
<p>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 <code>packed</code> 次元だけが指定されているので、<code>implicit_data_type</code> のルールにより、戻り値は 4 ビットの <code>logic</code> になります。</p>	<p>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 <code>implicit_data_type</code> のルールにより、戻り値は 1 ビットの <code>logic</code> 型となります。</p>
<p><code>data_in</code> にはデータタイプが指定されずに <code>packed</code> 次元だけが指定されているので、<code>implicit_data_type</code> のルールにより、<code>data_in</code> は 8 ビットの <code>logic</code> 型となります。</p>	

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ル名 (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しょう。

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。

まず、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。

SystemVerilog 規格のまとめ

- SystemVerilog 規格は、以下のような変遷を辿って来ています。

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ません。

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。

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。

The diagram shows a horizontal timeline of five yellow arrow-shaped boxes pointing right, representing the evolution of IEEE standards: IEEE Std 1800-2005, IEEE Std 1800-2009, IEEE Std 1800-2012, IEEE Std 1800-2017, and IEEE Std 1800-2023. Dotted lines connect these standards to the explanatory text blocks above and below.

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。

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。

なぜ SystemVerilog? Copyright 2024 © Artgraphics. All rights reserved. 5

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？

プロセス

- 答えはYesです。個々の連続代入文もプロセスです。しかも、シミュレーション実行中、常に、存在して実行しています。連続代入文は以下のように使用しますが、この文自身で単独のプロセスを表現します。

```
assign z = a > b;
```

- このプロセスは@(`a,b`)のイベント待ちですが、イベントが発生するとこの文が呼び出されるわけではありません。イベントが起こると、このプロセスはアクティブになり、右辺の計算を開始し結果が現在の`z`の値と異なれば、`z`に新しい値を代入します。その後、`z`に関するイベントが起きた事を通知します。そして、次回の@(`a,b`)のイベントを待ちます。
- 上記のように簡単な連続代入文では納得いかないと思えるので、もう少し高度な例を紹介します。例えば、以下のような連続代入文ではプロセスでないと実現できません。

```
assign #2.5ns sum = a + b;
```

- 2.5ns後に右辺の値を左辺に設定する仕事は、ファンクションのような呼び出しでは実現できません。

Copyright © 2025 Artgraphics. All rights reserved.

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ン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。

コンストラクタとdefault

- defaultを使用するとコンストラクタの定義を省略できる場合があります。LRMの例を紹介します。ベースクラスが以下のように定義されているとします。

```
class Base;
string name;
local int m_id;
function new(string name,output int id);
    this.name = name;
    id = m_id++;
endfunction
endclass
```

- extendsで引数を指定すると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るので、サブクラスのコンストラクタ内ではsuper.new()を呼び出してはいけません。

コンストラクタを指定する方法	defaultを利用した定義法
<pre>class A extends Base(default); int size; function new(int size,default); this.size = size; endfunction endclass</pre>	<pre>class B extends Base(default); endclass</pre>
<ul style="list-style-type: none"> ● サブクラスのコンストラクタには処理が必要なのでコンストラクタを定義しています。 ● しかし、extendsで引数を指定しているので、super.new(default)を呼び出す事はできません。 	<ul style="list-style-type: none"> ● サブクラスのコンストラクタに特別な処理がなければ、サブクラスではコンストラクタの指定を省略できます。

Copyright © 2025 Artgraphics. All rights reserved.

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。

```

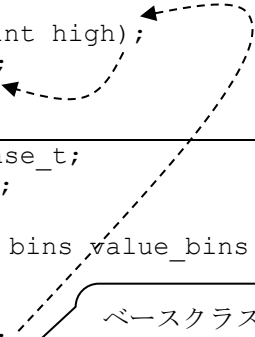
class base_t;
rand bit [2:0] a;
covergroup cg(int low,int high);
    coverpoint a { ignore_bins value = { [low:high] }; }
endgroup
function new(int low,int high);
    cg = new(low,high);
endfunction
endclass

```

```

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
<pre>case ({a,b,c}) 3'b110, 3'b101, 3'b011: z = 1'b1; default z = 1'b0; endcase</pre>	<pre>z = {a,b,c} inside {3'b110,3'b101,3'b011};</pre>
<pre>casex ({a,b,c}) 3'b11x, 3'b1x1, 3'bx11: z = 1'b1; default z = 1'b0; endcase</pre>	<pre>z = {a,b,c} inside {3'b11?,3'b1?1,3'b?11};</pre>

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

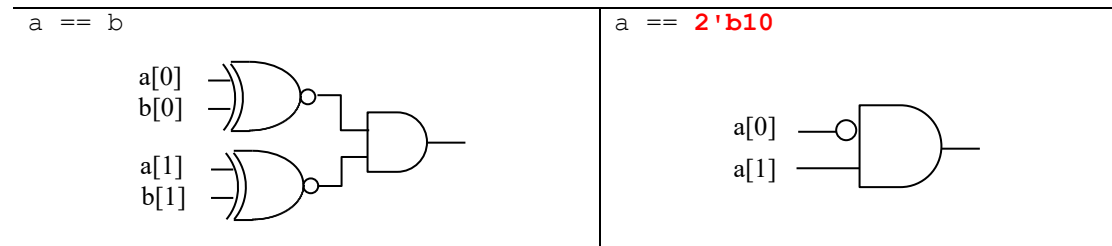
36 リテラル 2025.04.03

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。

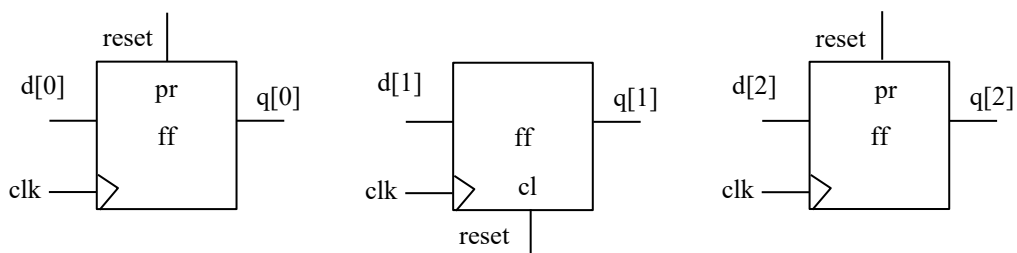


逆に、シーケンシャル回路の場合にはリテラルを使用すると複雑になります。以下の記述は、微妙に異なるフリップフロップを生成します。

```

module pdfff(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

```



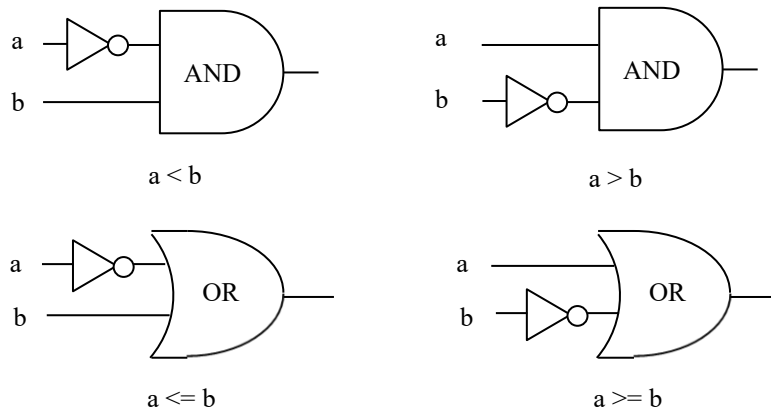
参考文献

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。

まず、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。

$$a \rightarrow b$$

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が。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` 等の全てのプロセスが終了するか待ち状態に入ってから、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きました。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 保守・拡張性 2025.08.08

設計分野でも検証分野でも拡張性・保守性を考慮する必要がありますが、SystemVerilog ではそれは可能です。

SystemVerilog のクラスを使用した開発では、クラスにパラメータを伴うのが常です。しかも、クラスのパラメータは時間が経過するにつれて変化していく可能性があります。そのような状況を考慮すると、パラメータの変化にできるだけ依存しない記述法が必要になります。以下は、そのような開発例です。

特別な考慮がされていない記述	保守性・拡張性を考慮した記述
<pre>class registry #(type T=int); // ... static function registry#(T) get(); static registry#(T) m_inst; if (m_inst == null) m_inst = new(); return m_inst; endfunction ... endclass</pre>	<pre>class registry #(type T=int); // ... static function type(this) get(); static type(this) m_inst; if (m_inst == null) m_inst = new(); return m_inst; endfunction ... endclass</pre>
<p>クラス内では、クラスタイプ自身を参照する事が頻繁に起こります。上記のように、クラスタイプが散乱していると、クラスのパラメータに追加・変更があると、修正に手間がかかります。</p>	<p>クラスタイプを type(this) で参照しているので、クラスのパラメータに変更が生じても get() メソッド内の記述に影響がありません。たとえ、クラス名が変更されても影響がありません。</p>

47 簡略記法 2025.08.24

SystemVerilog では、packed アレイであろうと unpacked アレイであろうと、便利な簡略記法があるので、同じような繰り返しの記述は恰好な適用例です。

以下の状況を仮定して、簡単な例を紹介します。


```
logic [7:0] a, z;
logic [3:0] c[5];
```

標準的な方法	簡略記法
<pre>z[6] = a[7]; z[5] = a[6]; z[4] = a[5]; z[3] = a[4];</pre>	<pre>z[6:3] = a[7:4];</pre>
<pre>z[7] = 1'b0; z[6] = a[6]; z[5] = a[6]&a[5]; z[4] = a[5]&a[4]; z[3] = a[4]&a[3]; z[2] = a[3]&a[2]; z[1] = a[2]&a[1]; z[0] = ~a[0];</pre>	<pre>z = {1'b0, a[6], a[6:2]&a[5:1], ~a[0]};</pre>
<pre>a[5]^a[4]^a[3]^a[2]</pre>	<pre>^a[5:2]</pre>
<pre>c = '{ 2, 2, 2, 2, 2 };</pre>	<pre>c = '{default:2};</pre>
	<pre>c = '{5{2}};</pre>
	<pre>foreach(c[i]) c[i] = 2;</pre>

48 for 文 2025.09.13

SystemVerilog の知識を得ると従来とは異なる記述法がある事に気が付く機会が多くなります。そのような例を示す典型的な記述を紹介します。

規則性のある SystemVerilog 記述には、for 文の機能を適用できる場合が多く存在します。下表左には良く知られているプライオリティコーディングが記されていますが、記述には規則性があるので for 文で書き直せます。

従来の方式によるプライオリティコーディング	for 文によるプライオリティコーディング
<pre> module priority_encoder(input [7:0] data, output logic [2:0] code); always @(data) if(data[7]) code = 7; else if(data[6]) code = 6; else if(data[5]) code = 5; else if(data[4]) code = 4; else if(data[3]) code = 3; else if(data[2]) code = 2; else if(data[1]) code = 1; else if(data[0]) code = 0; else code = 'x; endmodule </pre>	<pre> module priority_encoder(input [7:0] data, output logic [2:0] code); always @(data) begin code = 'x; for(int i = 7; i >= 0; i--)  end endmodule </pre>

右の記述が左の記述と同じ動作をするように右の四角内に記述を追加して下さい。ただし、論理合成可能であるかは問いません。

49 case/casex 2025.09.20

初心者は SystemVerilog の基本的な部分にしばしば疑問を持ちます。その際には、先輩から簡潔明瞭な回答が期待されます。

ある初心者は以下の記述における相違が良く分からないとします。

case 文	casex 文
<pre>module mux_case(input a,b,c,d, sa,sb,sc,output logic out); always @(a,b,c,d,sa,sb,sc) case ({sa,sb,sc}) 3'b1xx: out = a; 3'bx1x: out = b; 3'bxx1: out = c; default out = d; endcase endmodule</pre>	<pre>module mux_casex(input a,b,c,d, sa,sb,sc,output logic out); always @(a,b,c,d,sa,sb,sc) casex ({sa,sb,sc}) 3'b1xx: out = a; 3'bx1x: out = b; 3'bxx1: out = c; default out = d; endcase endmodule</pre>

- ① 初心者に、両者の相違を簡潔明瞭に説明して下さい。
- ② casex 文の記述に近い動作をするためには、case 文の記述をどのように書き直せば良いですか？

50 知識と技術 2025.09.25

Verilog でも SystemVerilog でも同じですが、知識を活用して技術を進歩させて行く必要があります。

簡単な例を使用して、技術の進歩が必要な点を明確にします。まず、非常に簡単なマルチプレクサのデザインを紹介します。

```
module design1(input a,b,c,d,[1:0] s,output logic z);
  always_comb
    if( s == 0 )
      z = a;
    else if( s == 1 )
      z = b;
    else if( s == 2 )
      z = c;
    else
      z = d;
endmodule
```

この記述の代わりに、「一つの代入文で同じ機能を実現させなさい」と新人が指示されたとします。すると、恐らく、以下のような記述を準備すると思われる。

```
module design2(input a,b,c,d,[1:0] s,output logic z);
  assign z = s == 0 ? a : s == 1 ? b : s == 2 ? c : d;
endmodule
```

しかし、?:オペレーターを使用しない簡単な方法もあります。

```
module design3(input a,b,c,d,[1:0] s,output logic z);
  assign z = ██████████;
endmodule
```

ところが、design2 および design3 が 4:1 マルチプレクサに合成されるとは限りません。3 個の 2:1 マルチプレクサが合成される可能性はかなり高くなります。したがって、その他の記述法が望ましいと言えます。更に別の方法もあります。以下の場合にはネットを必要としますが、4:1 マルチプレクサに合成される可能性は非常に高くなります。

```
module design4(input a,b,c,d,[1:0] s,output logic z);
  wire ██████████;
  assign z = ██████████;
endmodule
```

51 キュー 2025.10.04

SystemVerilog のキューは優れた特性を持っているので、どのようなデータ構造で実装されるか気になる所です。

SystemVerilog の LRM は、キューを以下のように特徴づけています。

LRM 169 ページ

A queue is a variable-size unpacked array that supports constant-time insertion and removal at the beginning or the end of the array as well as constant-time access to all its elements

性能をまとめると以下ようになります。

SystemVerilog のキューの特徴

特性	オペレーション	処理時間
1	キューの先頭での挿入、削除	$O(1)$
2	キューの最後での挿入、削除	$O(1)$
3	キューの要素へのインデックスでのアクセス	$O(1)$

標準的なリスト構造では特性 3 を満たしません。どのようなデータ構造であれば LRM の仕様を満たせるかを考えてみると面白いです。

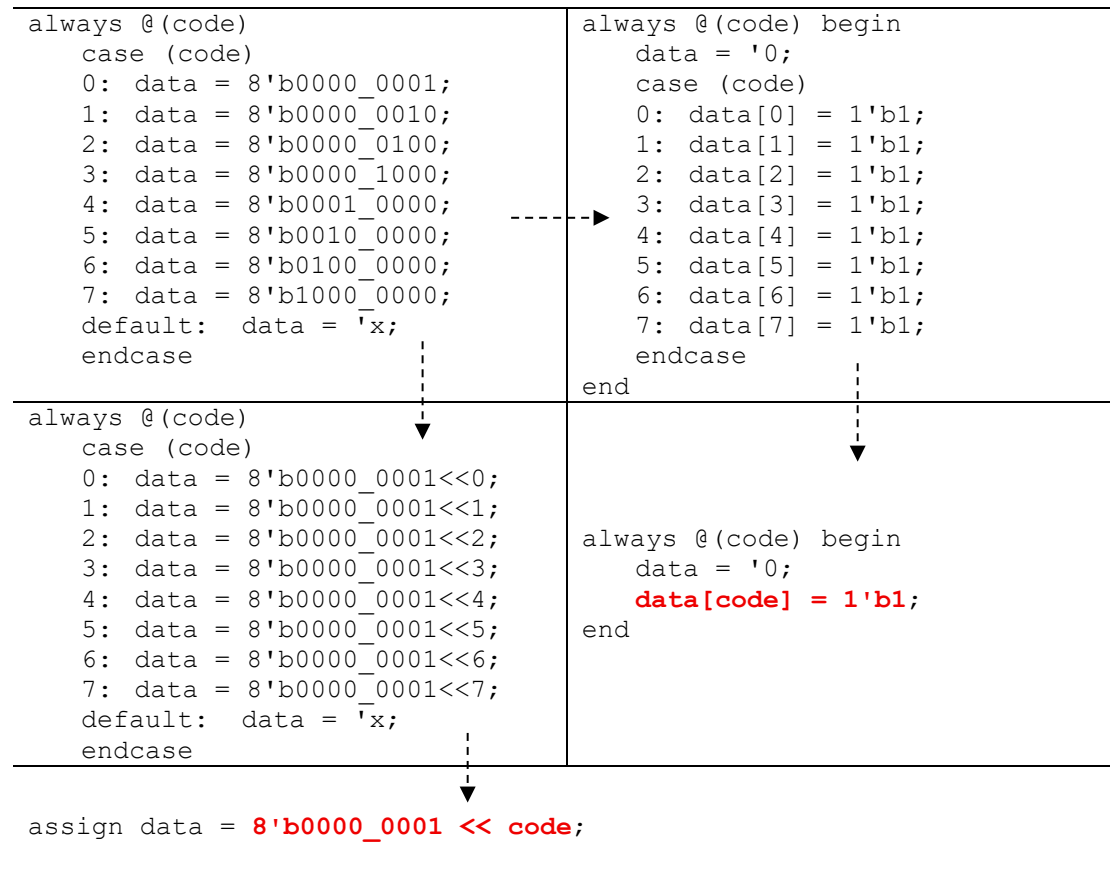
52 SystemVerilog 記述 2025.10.23

ハードウェア記述には規則性があるので、どのように変化をとらえるかで表現法が異なります。VerilogでもSystemVerilogでも同じですが、多くの場合にはビットセレクトまたはシフト演算等の簡単な記述に落ち着きます。

以下のように信号が定義されているとします。

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

すると、以下のように落ち着く先は簡単な記述になります。だからこそ、論理合成可能であると言えます。



参考文献

SystemVerilogによるモデリングと論理合成、アートグラフィックス 2025.

53 サブクラス 2025.10.27

少し高度な SystemVerilog の話題になりますが、サブクラスを定義する際、`extends` で指定するベースクラスに引数を添えられます。すると、自動的に、`super.new()` が呼び出されます。したがって、サブクラス内で `super.new()` を呼び出す必要はありません。

以下の記述例では、`extends` する際に引数を指定しています。そのため、サブクラスのコンストラクタ内に `super.new(new)` が自動生成されます。したがって、サブクラスのコンストラクタ内では `super.new()` の呼び出しをする必要はありません。もし呼び出しを記述するとコンパイルエラーになります。

```
class Base;
string    m_name;
logic [3:0] m_state;
function new(string name="name", logic [3:0] state);
    m_name = name;
    m_state = state;
endfunction
endclass
```

extends する際に引数を指定しているので、自動的に `super.new(default)` が生成される

```
class A extends Base(default);
int    size;
function new(int size, default);
    this.size = size;
endfunction
endclass
```

`super.new(default)` の呼び出しを記述する事はできない

54 乱数発生 2025.10.29

linear feedback shift registers と呼ばれる乱数発生アルゴリズムがあります。N ビットレジスタを右に 1 ビットシフトし、空いた MSB を他のビットの論理演算の結果で埋めるアルゴリズムです。論理演算を工夫すると長さ $2^{*}N-1$ の乱数列を生成できます。SystemVerilog でアルゴリズムを紹介します。

SystemVerilog では簡単に試せます。N=4 とし論理演算を LSBs の 2 ビットの XOR とします。すると、1~15 の数から構成される長さ 15 の乱数列を生成できます。XOR を取るのでレジスタの初期値は 0 以外でなければなりません。以下のコードでは初期値を '1 にしています。

```
module LFSR #(NBITS=4,TAP=1)
  (input clk,reset,output logic [NBITS-1:0] q);
always @(posedge clk,posedge reset)
  if( reset )
    q <= '1;
  else
    q <= {q[TAP]^q[0],q[NBITS-1:1]};
endmodule
```

実行すると以下のように、長さ 15 の乱数列を生成できます。

```
( 1) @ 10: 0111 ( 7)
( 2) @ 30: 0011 ( 3)
( 3) @ 50: 0001 ( 1)
( 4) @ 70: 1000 ( 8)
( 5) @ 90: 0100 ( 4)
( 6) @110: 0010 ( 2)
( 7) @130: 1001 ( 9)
( 8) @150: 1100 (12)
( 9) @170: 0110 ( 6)
(10) @190: 1011 (11)
(11) @210: 0101 ( 5)
(12) @230: 1010 (10)
(13) @250: 1101 (13)
(14) @270: 1110 (14)
(15) @290: 1111 (15)
(16) @310: 0111 ( 7)
(17) @330: 0011 ( 3)
...

```

} 長さ 15 の乱数列

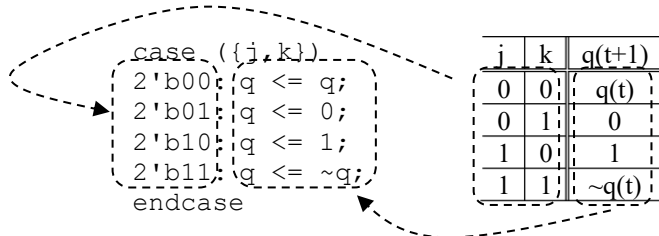
←----- ここから繰り返す

異なる N の場合には、LSBs の 2 ビットでは成功しません。例えば、N=5 の場合、q[0] と q[1] の XOR では周期は 21 ですが、q[0] と q[2] の XOR では周期が 31 となります。N=31 の場合、q[0] と q[4]、q[7]、q[8]、q[14]、q[19]、q[25]、q[26]、q[29] の何れかが良いとされています。

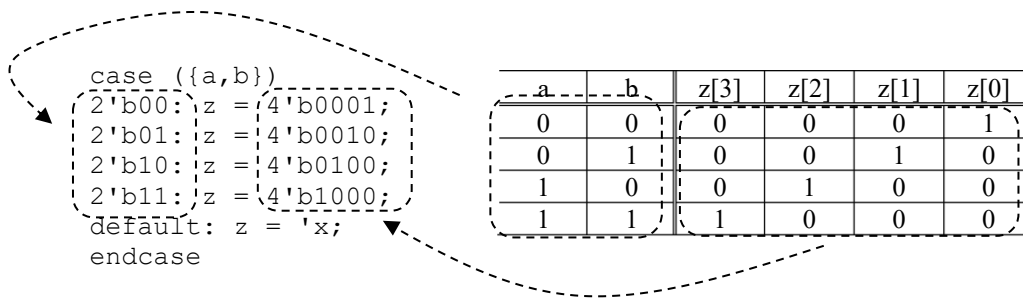
55 真理値表と case 文 2025.11.03

真理値表といえば、SystemVerilog では user-defined primitive (UDP)を思い出すかも知れませんが、UDP は時代遅れであるだけでなく使用制限があります。したがって、UDP に代わる手段が必要になりますが、実は、case 文がその役割をします。

例えば、jk-フリップフロップの真理値表は下図の右のように与えられますが、入力部分を case 文の項目、出力部分を case 文の実行文に指定すれば、真理値表を SystemVerilog で記述できます。



デコーダー等は、最も適切な例です。



casex 文を使用すると do-not-care を使用できるのでコンパクトな表現が可能になります。したがって、ハードウェア記述において真理値表は極めて有用な手段と言えます。SystemVerilog の便利な機能は下記の文献に詳しく解説してあります。

[1] SystemVerilog 超入門、共立出版 2023.

[2] SystemVerilog 入門、共立出版 2020.

56 case+inside ⇐ casex 2025.11.05

SystemVerilog の case 文に inside オペレータを指定すると、右オペランドにワイルドカードが許されるので、case 文は casex の機能に近くなります。

case	case+inside ⇐ casex
<pre> module design1(input a,b,c,d,sa,sb,sc, output logic out); always_comb case ({sa,sb,sc}) 3'b1xx: out = a; 3'bx1x: out = b; 3'bxx1: out = c; default out = d; endcase endmodule </pre>	<pre> module design2(input a,b,c,d,sa,sb,sc, output logic out); always_comb case ({sa,sb,sc}) inside 3'b1xx: out = a; 3'bx1x: out = b; 3'bxx1: out = c; default out = d; endcase endmodule </pre>
<p>case 文では、do-not-care を使用できないので、{sa,sb,sc}=3'b100 でも out は a に設定されません。</p>	<p>case 文にも関わらず、3'b1xx、3'bx1x、3'bxx1 に現れる x は do-not-care として扱われます。{sa,sb,sc}=3'b100 であると out に a が設定されます。</p>

inside オペレータは下記の文献に詳しく解説されています。

参考文献

- [1] SystemVerilog 超入門、共立出版 2023.
- [2] SystemVerilog 入門、共立出版 2020.

57 case+inside⇔casex 2025.11.07

SystemVerilog では、case 文に inside オペレータを指定できますが、casex/casez には指定できません。何故ですか？良い問題なので、是非、試して下さい。

解答

LRM 320 ページ

```

case_statement ::=
  [ unique_priority ] case_keyword ( case_expression )
  case_item { case_item } endcase
  | [ unique_priority ] case_keyword ( case_expression ) matches
  case_pattern_item { case_pattern_item } endcase
  | [ unique_priority ] case ( case_expression ) inside
  case_inside_item { case_inside_item } endcase
case_keyword ::= case | casez | casex
case_expression ::= expression
...

```

- ① 上記のシンタックスにあるように casex/casez では inside オペレータを指定できません。そもそも、指定する必要がありません。例えば、3'b1xx は[4:7]、3'bx1x は {2,3,6,7}を示します。つまり、先天的に casex/casez では範囲指定やリスト指定が可能です。
- ② 一方、case 文では値の範囲指定をできないため、記述が冗長になります。そのため、inside オペレータの機能が必要になります。例えば、4 ビットの 4 の倍数を指定する際、4'bxx00 と簡単に記述できるようになります。あるいは、3 ビットの奇数を 3'bxx1 で表現できるようになります。

□

inside オペレータは下記の文献に詳しく解説されています。

参考文献

- [1] SystemVerilog 超入門、共立出版 2023.
- [2] SystemVerilog 入門、共立出版 2020.

58 サブクラスでの制約定義 2025.11.13

SystemVerilog のサブクラスで制約を定義する場合、`:initial`、または、`:extends` を指定する習慣を付けておくと良いです。

サブクラスで新たに定義する制約には **`:initial`** を指定しておくと良いです。もし、将来、ベースクラスにその名称を持つ制約が追加されると、コンパイラーがエラーを発行してくれるので、ベースクラスかサブクラスの何れかの制約名を変更すれば問題を解決できます。

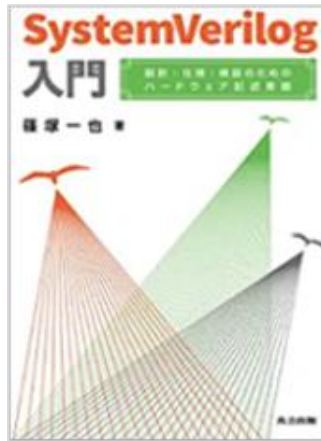
逆に、ベースクラスの制約をサブクラスで書き換える際には、**`:extends`** を指定しておくと良いです。例えば、ベースクラスにその名称を持つ制約が存在しなければコンパイルエラーが出ます。**`:extends`** の指定がないとエラーが出ないため、悶々とデバッグの日々を過ごさなければなりません。これらの識別子を以下のように使用できます。

```
class sub_t extends base_t;
  constraint :initial C1 {...}
  constraint :extends C2 {...}
  ...
endclass
```

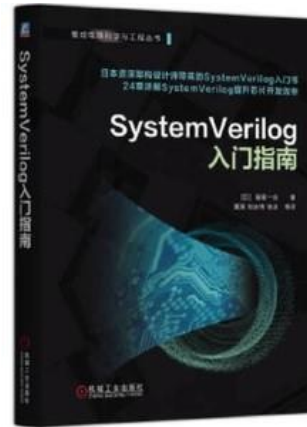
59 SystemVerilog 入門の中国語版 2025.11.15

『SystemVerilog 入門』が中国語でも読めるようになりました。

共立出版社から刊行されている『SystemVerilog 入門』が、中国国営の機械工業出版社により中国語に翻訳され『SystemVerilog 入門指南』として刊行されました。



日本語版：共立出版社



中国語版：機械工業出版社

基礎知識を確実にするのは技術者にとって何よりも大切な事です。『SystemVerilog 入門』を読むと世界で通用する知識・技術を習得できます。初心者には『SystemVerilog 超入門』をお勧めします。

60 SystemVerilog 2025.12.04

SystemVerilog の改訂も落ち着き言語仕様は成熟した域に到達しました。しかし、使用面での画期的な進歩は見られないように思えます。SystemVerilog が IEEE 標準規格となってから 20 年経過した現在、言語仕様に眠っている機能や新たな活用法を模索する時期であると思います。

SystemVerilog によるモデリングと論理合成

- 論理合成の原理と SystemVerilog による現代的なモデリング手法を学べる面白い本です。
- RTL 記述から合成される回路構成を予測する知識・技術を習得できます。
- ブール代数の基礎知識だけで論理の最適化手法を学べます。
- 的確な RTL モデリング知識・技術を身に付けられます。従来の手法では思いにもよらなかった効果的なモデリング手法が解説されています。
- Gray コードの特徴が詳しく解説されています。更に、ブール代数の解説及び FSM の厳密な定義もあるので技術者のみならず学生諸氏にもお勧めします。



61 SystemVerilog 2025.12.29

SystemVerilog には記述法が沢山あります。特に、for 文はモデリングに活用できるケースが多くあります。

時間的な余裕があれば、下記の問題を考えてみると面白いです。

RTL モデリング	問題
<pre> module design(input a, b, c, d, e, f, g, h, [7:0] data, output logic z); always_comb if(data[7]) z = a; else if(data[6]) z = b; else if(data[5]) z = c; else if(data[4]) z = d; else if(data[3]) z = e; else if(data[2]) z = f; else if(data[1]) z = g; else if(data[0]) z = h; else z = 1'bx; endmodule </pre>	<p>① for 文を使用して左の動作記述を書き直して下さい。</p> <p>② case 文を使用して書き直して下さい。ただし、casez、casex、および inside を使用しないで下さい。つまり、ワイルカードの使用を禁止します。</p>

この話題に興味を持つ方には下記の文献を勧めます。

参考文献

SystemVerilog によるモデリングと論理合成、共立出版 2026.

62 Gray コード 2025.12.31

SystemVerilog の case 文でビットを並べる場合には、Gray コード順に並べると最適化し易くなります。

Gray コードは、距離が 1 となるようにビット定数を順に整列する作用があります。この特性を利用すれば、間違いなく最適化した整数を並べられます。

<pre>always_comb case (s) 3'b011, 3'b101, 3'b110, 3'b111: z = 1'b1; default z = 1'b0; endcase</pre>	Gray コード順 ⇒	[<pre>3'b011 3'b110 3'b111 3'b101</pre>]	⇒	最適化	[<pre>3'bx11 3'b11x 3'b1x1</pre>]
		3'b011 と 3'b111 は MSB を反転した関係にある			↓ 書き換え
					<pre>always_comb casex (s) 3'bx11, 3'b1x1, 3'b11x: z = 1'b1; default z = 1'b0; endcase</pre>

63 実数型のインクリメントとデクリメント 2026.01.12

SystemVerilog では `a++`、`++a`、`a--`、`--a` のような記述が許されますが、もし `a` が実数型であると、どのような動作をすると思いますか？

そうです。1 の代わりに 1.0 が使用されて演算が実行されます (LRM 275 ページ)。

64 オペレータの結合法則 2026.01.12

言語にオペレータが定義されていると、オペレータの優先順位が必ず定義されます。そして、同じ優先順位を持つオペレータが同時に使用されると、結合法則が適用されて矛盾が起こらないようにします。SystemVerilog において、オペレータの結合法則を正しく解釈しておくことは重要です。

SystemVerilog LRM では、オペレータの結合法則を以下のように定義しています。この定義は、同じ優先順位をもつオペレータに対しての規約ですが、理解し難いと思います。

LRM 273 ページ

All operators shall associate left to right with the exception of the conditional (`?:`), implication (`->`), and equivalence (`<->`) operators, which shall associate right to left. Associativity refers to the order in which the operators having the same precedence are evaluated.

以下の式を考えてみます。問題は、 $(9-5)+2$ なのか、それとも $9-(5+2)$ を意味するからです。

|| $9-5+2$

リテラル 5 の両側にオペレータ (-) とオペレータ (+) があり、それらの優先順位は同じです。したがって、上記のルールが適用されます。ルールによれば、左から結合されます。つまり、リテラル 5 は、左側のオペレータ (-) に結合されます。厳密に言えば、左側のオペレータ (-) がリテラル 5 を結合します。したがって、式は以下のように評価されます。

|| $(9-5)+2$

参考

算術式と異なりブール代数には同じ優先順位を持つオペレータは存在しません。従って、上記の規約は不必要です。驚く事に、ブール代数の公理からブール代数では結合法則が成立する事を証明できます。詳細は、下記文献を参照して下さい。皆さんが設計している回路はブール代数のインスタンスなので、自然に結合法則が成立しています。従って、回路の出力に曖昧性はありません。ブール代数の性質を再確認しておくに役立ちます。

SystemVerilog によるモデリングと論理合成、共立出版 2026.

□

65 inside 2026.01.20

昔は、真理値表といえばUDPを連想しましたが、SystemVerilogの出現以来、その考えは変わりました。少なくとも組み合わせ回路の真理値表であれば、inside および case 文で対応できます。

例えば、出力が1つの真理値表であれば inside オペレータで十分です。このオペレータにはワイルドカードを使えるので、式を簡潔に表現できます。手順は以下の通りです。

- 出力が1となる入力の組み合わせを集める
- Grayコード順に並べ替える
- 連続する入力の組み合わせは最適化できるのでワイルドカードで表現する
- inside オペレータで式を表現する

s	a	b	z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

001]
 011] --> 0?1 --> z = {s,a,b} inside {3'b0?1,3'b11?}
 110]
 111]

しかし、この例の場合には、もう少し簡単な解法があります。以下のように、真理値表は上下に分割できます。上半分は $z=b$ 、下半分は $z=a$ です。この事実から、真理値表が 2:1 マルチプレクサを表現している事が分かります。このような分析力を養うためには下記文献をすすめます。

s	a	b	z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

z=b
 z=a

参考文献

SystemVerilog によるモデリングと論理合成、共立出版 2026.

66 attribute_instance 2026.02.02

ソースコードのコメントは、記述者及び読む人への注釈ですが、コンパイルすると消えてしまいます。コメントと同様にテキストを付加する機能としてアノテーションがあります。この機能は、言語をサポートしているツールへ引き渡される情報ですが SystemVerilog にもあります。

言語やアプリケーションが進化するとアノテーションが重要な役割を果たしてきます。コメントと異なり、アノテーションは言語を解釈して実行するアプリケーションに引き渡されるテキスト情報です。テキストであるため、殆ど無制限で情報をアプリケーションに引き渡されます。SystemVerilog のアノテーションは以下のシンタックスを持ちます。

```
attribute_instance ::= (* attr_spec { , attr_spec } *)
attr_spec ::= attr_name [ = constant_expression ]
attr_name ::= identifier
```

宣言文、実行文、オペレータ等のアノテーションできます。アノテーションは、対象となる言語要素の左または右に添えます。宣言文や実行文では左、オペレータやファンクションの呼び出しでは右側にアノテーションを添えます。

アノテーションの使用例	説明
(* optimize_power=1 *) module mod1 (<port_list>);	モジュールは宣言文なので左にアノテーションします。
(* fsm_state=1 *) logic [3:0] state1;	変数やネットの宣言では左にアノテーションをします。
a = b + (* mode = "cla" *) c;	オペレータは、右側にアノテーションをします。
a = add (* mode = "cla" *) (b, c);	ファンクションの呼び出しでは、右側にアノテーションをします。
(* full_case=1 *) (* parallel_case=1 *) case (a) <rest of case statement>	複数のアノテーションを指定できます。

アノテーションを考慮している SystemVerilog ツールがあれば、活用すると良いです。良く知られているアノテーション機能は、恐らく、Verilog-AMS の機能であると思います。以下のようにアノテーションすると Spice 系のシミュレータにアノテーションが引き渡されます。

```
(* desc="gate-source capacitance", units="F" *)
real cgs;
```

シミュレータは、アノテーションを使用して以下のようにプリントします。

```
cgs = 4.21e-15 F gate-source capacitance
```

アノテーションは、ツールに知識を供給する重要な機能なので、今後は一層活用されて行くと思われます。昔は、コメントでツールの動作を制御していましたが、現在ではアノテーションによりツールを制御する方式が主流です。

67 計算オペレータ 2026.02.07

時として、SystemVerilog/Verilog での 4 値の取り扱いが面倒です。例えば、 $a==b$ において、両辺の何れかが $1'bx$ または $1'bz$ を含んでいれば、結果は $1'bx$ となります。 $1'bx$ は偽と判断されるので、期待する結果を得られない場合が多々あります。

しかし、特定の処理においては、計算オペレータ (reduction operators) を活用すると、すこぶる簡潔な表現が可能です。SystemVerilog の基礎知識の重要性を紹介します。以下のように 8 ビットの変数 a が定義されているとします。

```
logic [7:0] a;
```

この変数には、 $1'bx$ 、および $1'bz$ が含まれている可能性があるとして、下記の質問を考えますが、 a は 2 値ではないので注意が必要です。例えば、①を解くのに $a!=0$ と書くと正しくありません。

-
- ① 変数 a に $1'b1$ が含まれている条件を簡潔に表現して下さい。
 - ② 変数 a に $1'b0$ が含まれている条件を簡潔に表現して下さい。
-

この問題を解くために、for 文や foreach 文を使用する必要はありません。以下のように簡単な解法があります。

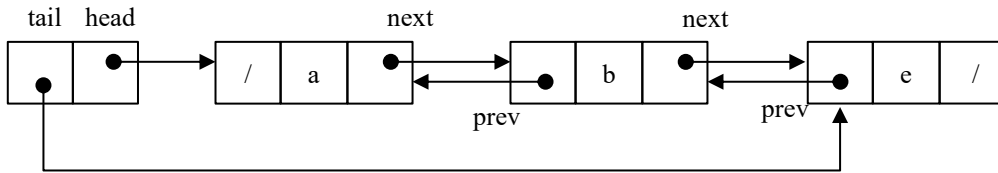
-
- ① $|a==1'b1$ 、あるいは、簡単に $|a$ でも良いです。
 - ② $&a==1'b0$ 、あるいは、 $!(\&a)$ と書いても良いです。
-

ビット演算において $1'b0$ と $1'b1$ は特別な力を持っています。 $1'b0$ はビット演算 ($\&$) に関して制御機能を持ち、 $1'b1$ はビット演算 ($|$) に関して制御機能を持ちます。

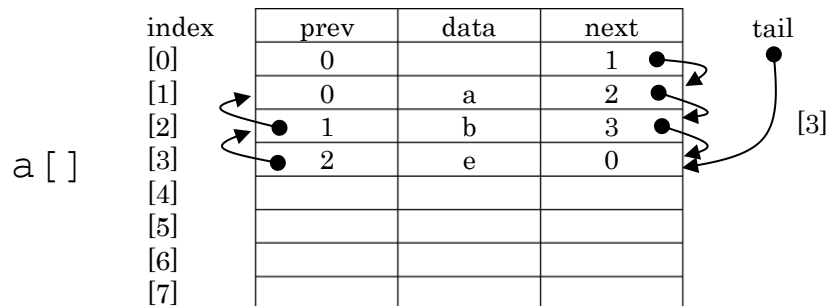
68 ダイナミックアレイ 2026.02.13

SystemVerilog でデータ構造を管理する場合には、ダイナミックアレイを使用すると効率的です。というのは、ダイナミックアレイは現在の内容を保持しながらアレイを拡張できるからです。

仮に、以下のようにオブジェクトを双方向リストで管理するとします。



SystemVerilog ではリスト管理をアレイで賄うと効率が良くなります。例えば、以下のように管理できます。ここでは、アレイの[0]はリストヘッドの役割をさせます。また、N を現在のアレイの大きさとしします。ここには示していませんが、削除された項目を再利用する仕組みも簡単に実装できます。



アレイを[0]から順に使用して行くので、[N-1]まで使い尽くすとアレイには空き領域がなくなります。しかも、[0]~[N-1]はリスト構造のデータが詰まっているので、保存しながらアレイを拡張しなければなりません。しかし、SystemVerilog のダイナミックアレイでは、アレイ内容の保存はシミュレータが保証してくれるので、以下のようにすると、現在の内容を保存しながらアレイの大きさを2倍にできます。

```
a = new[a.size()*2](a);
```

僅か 1 行で領域を拡張できます。リスト管理をアレイインデックスで行えば、アレイの領域が変化しても影響を受ける事がないのでオブジェクト管理が容易です。検証では、ダイナミックアレイを利用すると効率的な処理を実現できます。

69 引数としてのファンクション 2026.02.25

SystemVerilog には C/C++ に類似した機能が多くみられます。例えば、`$display` システムタスクの書式は C/C++ とほぼ同じです。しかし、C/C++ で便利な機能でも SystemVerilog には存在しないものもあります。

SystemVerilog において微分や積分の計算をする場合には関数を引数に指定したくなります。C/C++ では以下のようにできます。中心差分法により $f(x)$ の微分係数を計算しています。

```
typedef double (*t_function) (double x);
double ddx(t_function f, double x)
{
    t_double h = 0.001;
    return (f(x+h)-f(x-h))/(2*h);
}
```

C/C++ に近い実装法としては、SystemVerilog ではクラスを活用する方法が考えられます。以下の `math_t` クラスでは、関数 $f(x)$ の微分係数を求める関数 `ddx(x)` を定義しています。ここで、関数 $f(x)$ は `virtual` メソッドでありサブクラスが定義できるようにしておきます（下表参照）。

微分積分を計算するベースクラス	$f(x) = x^3$
<pre>virtual class math_t; real mH = 0.001; pure virtual function real f(real x); function real ddx(real x); ddx = (f(x+mH)-f(x-mH))/(2*mH); endfunction endclass</pre>	<pre>class x3_t extends math_t; function real f(real x); f = x**3; endfunction endclass</pre>

こうしておくと、 $f(x) = x^3$ の微分 $\frac{df}{dx}$ を以下のように計算できます。

```
x3_t    x3;
x3 = new;
$display("%g", x3.ddx(3.01));
```

$f(x) = x^3$ の $x = 3.01$ における微分係数は 27.1803 と計算されますが、これは完璧な値です。次回は、積分機能を追加します。

70 積分 2026.02.26

SystemVerilog での求積法を紹介します。前回定義した数学クラスに機能を追加します。

積分を求めるファンクション `integral` を `math_t` クラスに追加します。Simpson 法により面積を求める手法はかなり正確なので採用しています。

```
virtual class math_t;
real mH = 0.001;

pure virtual function real f(real x);

function real ddx(real x);
    ddx = (f(x+mH)-f(x-mH))/(2*mH);
endfunction

function real integral(real a,b,int divisions);
real w;
    w = (b-a)/divisions;
    integral = 0;
    for( int i = 0; i < divisions; i++ ) begin
        integral += w*(f(a+i*w)+4*f(a+w/2+i*w)+f(a+(i+1)*w))/6;
    end
endfunction
endclass
```

定義した積分を使用して $\int_1^2 \frac{1}{x} dx$ を求めるためにサブクラスを定義します。

```
class xinverse_t extends math_t;
function real f(real x);
    f = 1/x;
endfunction
endclass
```

] ----- $f(x) = \frac{1}{x}$ の定義

このように定義した関数の積分を以下のように求められます。

```
xinverse_t      inv;
inv = new;
$display("integral of 1/x over [1,2] = %g", inv.integral(1,2,10));
```

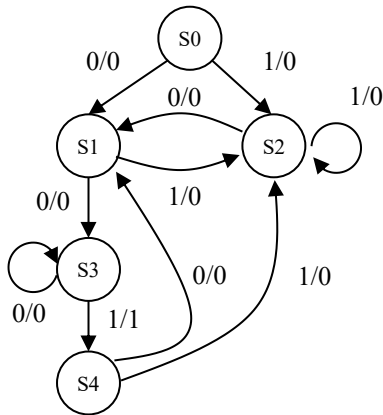
以下の結果を得ます。この求積法は区間[1,2]を 10 分割して計算していますが、結果は非常に正確です。ここで紹介した数学クラスを使用すると各種の実験ができます。

$$\int_1^2 \frac{1}{x} dx = 0.693147$$

71 FSM 2026.03.12

Mealy FSM でデザインする場合、状態遷移図と共に状態遷移表を描くと FSM の最適化を導ける場合が多くなります。具体的な最適化例と SystemVerilog による実装を紹介します。

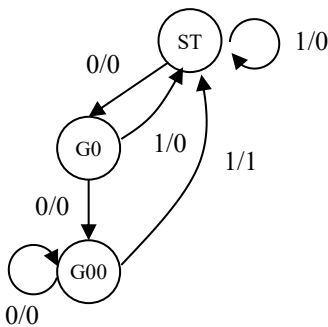
1 ビット単位で入力する Mealy FSM を仮定します。入力パターンとして 001 が出現すると 1 を出力し、それ以外は 0 を出力するとします。例えば、1001 を入力すると 0001 が出力になります。状態遷移図を描くと(a)のようになります。しかし、状態遷移表(b)を観察すると、S0、S2、S4 の動作は全く同じである事が分かります（つまり、同じ入力に対して次の状態と出力が一致します）。したがって、これらのノードを結合できます。結果として(c)を得ます。ST は start、G0 は got 0、G00 は got 00 の意味です。状態遷移表(b)を描かないと、このような最適化を導き難いと思われます。SystemVerilog の記述は(d)のようになります。



(a) 状態遷移図

(b) 状態遷移表

state	input	next state	output
S0	0	S1	0
	1	S2	0
S1	0	S3	0
	1	S2	0
S2	0	S1	0
	1	S2	0
S3	0	S3	0
	1	S4	1
S4	0	S1	0
	1	S2	0



(c) 最適化した状態遷移図

(d) SystemVerilog 実装例

```

module fsm_001(input clk,reset,in_bit,output logic out_bit);
state_e    state, next_state;

always @(posedge clk,posedge reset)
  if( reset )
    state <= START;
  else
    state <= next_state;
always @(state,in_bit) begin
  out_bit = (state == GOT_00) && (in_bit == 1);
  case (state)
  START:  next_state = in_bit==1 ? START : GOT_0;
  GOT_0,
  GOT_00: next_state = in_bit==1 ? START : GOT_00;
  endcase
end
endmodule
  
```

FSM に関して、下記の文献に詳しい解説があります。

参考文献

[1] SystemVerilog によるモデリングと論理合成、共立出版 2026.

72 HDL 2026.03.28

一般のプログラミング言語を使用する場合には、データ構造やアルゴリズムを設計する事から始めますが、HDL となると異次元の世界となりがちです。しかし、良く考察してみるとその必要がない事が多く見受けられます。SystemVerilog で簡単な例を示します。

値を入れ替える必要性が発生する場合があります。良く知られている例は、バイトをスワップする処理です。SystemVerilog では右記のようにするのが一般的です。この処理は、非常に一般的であり多くの応用例が考えられます。例えば、アレイに適用すると以下ようになります。これは、アレイを逆順に並べ替え変えます。

```
function void swap(inout int n[]);
int i, j, tmp;
i = 0;
j = n.size()-1;
while( i < j ) begin
    tmp = n[i];
    n[i] = n[j];
    n[j] = tmp;
    i++;
    j--;
end
endfunction
```

} アレイ要素のスワップ

しかし、SystemVerilog では、右記のように 1 行で記述できるので、上記の記述には利点がありません。もしアレイ n が packed アレイである場合には、上記の記述はビットを逆順に並べる処理となりますが、この場合にも右記のように SystemVerilog では 1 行で簡単に記述できます。

```
n.reverse();
logic [7:0] a, b;
b = {<<{a}};
```

以上に紹介したように、SystemVerilog では、アレイ操作に必要な機能は豊富なので、特別なアルゴリズムを必要とする場合は極めてまれであると言えます。

73 final class 2026.04.02

SystemVerilog でクラスを設計する場合、サブクラスの定義を禁止したい際には、final クラスとして宣言しておくが良いです。

サブクラスを定義して欲しくない場合は度々発生します。例えば、virtual メソッドを書き換えられなくしたい場合があります。サブクラスの定義を禁止するには、キーワード class の直後に :final を付け加えるだけで済みます。例えば、以下のようにして final クラスを定義できます。

```
class :final TopPacket extends LinkedPacket;
...
endclass
```

こうすると、TopPacket のサブクラスを定義できません。以下の例では、コンパイルエラーが発行されます。

```
class BrokenPacket extends TopPacket; // ILLEGAL
...
endclass
```

この機能により、std::process クラスは以下のように定義されています。

```
class :final process;
typedef enum {FINISHED, RUNNING, WAITING, SUSPENDED, KILLED} state;
static function process self();
function state status();
function void kill();
task await();
function void suspend();
function void resume();
function void srandom(int seed);
function string get_randstate();
function void set_randstate(string state);
endclass
```

SystemVerilog では、process クラスのサブクラスを定義できないようにするために、このように process クラスが定義されています。

74 final method 2026.04.03

SystemVerilog の `final` クラスは便利ですが、制限が強いため、一般的には、メソッド毎に `final` を適用する方が現実的です。この手法は、開発するコードの品質を高める効果をもたらします。

クラスのメソッド毎に制限を指定すると予期しない間違いを未然に防ぐ効果があります。以下の記述には、潜在的な問題があります。

```
class base_t;
virtual function void print(string msg);
...
endfunction
endclass
class sub_t extends base_t;
function void print(int msg);
...
endfunction
endclass
```

ベースクラスとサブクラスに同じ名称の `print` が定義されています。ところが、引数の型が異なるため、`sub_t::print` は、ベースクラスの `print` メソッドを別の目的に置き換えています。しかし、上記の記述にエラーは出ません。このような、間違いを未然に防ぐには、以下のようにします。

```
class sub_t extends base_t;
function :initial void print(int msg); // ERROR
...
```

ベースクラスの `virtual` メソッドが無効にされるため、コンパイラーがエラーを出します。また、以下のようにしても、やはりコンパイラーがエラーを発行してくれます。この場合には、ベースクラスの `print` メソッドの引数と型が一致しないというエラーが出ます。

```
class sub_t extends base_t;
function :extends void print(int msg);
...
```

ベースクラスの `print` メソッドを拡張できないようにするには以下のようにします。

```
class base_t;
virtual function :final void print(string msg);
...
```

参考文献

SystemVerilog IEEE Std 1800-2023 の概要、アートグラフィックス 2024.

この文献を共立出版社から販売されている『SystemVerilog 入門』、または『SystemVerilog 超入門』の書籍ウェブサイトから無償でダウンロードできます。

75 実数型のランダム変数 2026.04.10

SystemVerilog では、実数型変数もランダム変数に指定できます。

例えば、以下のように実数型変数をランダム変数に指定できます。

```
class sample_t;
rand logic [7:0] a;
rand real r;
constraint C1 {
    a inside {[0:7]};
    r > 0.0 && r < 2.0;
}
endclass
```

この場合には、以下のように `randomize` メソッドを使用して乱数を発生させます。

```
module test;
sample_t sample;

initial begin
    sample = new;
    repeat( 16 ) begin
        assert( sample.randomize() );
        $display("a = %0d, r = %-8g", sample.a, sample.r);
    end
end
endmodule
```

76 ファンクショナルカバレッジ 2026.04 12

ファンクショナルカバレッジを利用する事により広範囲の検証が促されます。結果としてデザイン上の問題を早期に解消する機会に恵まれます。簡単な SystemVerilog 例で効果を紹介し

ます。
4 ビット符号付き整数の符号を反転する回路のデザイン（下表の左）を検証します。実は、デザインに間違いがあります。検証では、主要な値を用いたテストケースが行われたかを確認するためにファンクショナルカバレッジを適用します（下表の右）。

符号を反転する回路（正しくない）	テストデータを生成するクラス
<pre>module complement (input signed [3:0] N, output signed [3:0] minusN); assign minusN = ~N+1'b1; endmodule</pre>	<pre>class item_t; rand logic signed [3:0] data; constraint C { data == -8 data == 7 data inside {[-2:2]}; } covergroup cg; coverpoint data { bins small_bin = {-8}; bins medium_bin = {[-2:2]}; bins large_bin = {7}; } endgroup function new; cg = new; endfunction endclass</pre>

テストベンチを右のように準備して実行してみると、100%のカバレッジを得られます。しかし、以下に示すように、回路の出力が正しくないケースが出てきます。

	N	-N
...		
@ 30:	-1	1
@ 40:	-8	-8
@ 50:	7	-7
...		

```
module test;
  logic signed [3:0] N, minusN;
  item_t item;

  complement DUT(.*);
  initial begin
    $display("      N -N");
    item = new;
    repeat(8) begin
      #10;
      assert (item.randomize());
      N = item.data;
      item.cg.sample();
    end
  end
  initial forever @(N) #0
    $display("@%3t: %2d %2d", $time, N, minusN);
endmodule
```

small_binにおけるテストケースで回路の問題点を見出しています。もし、このビンがカバーされていなければ、回路の問題は表面化しません。つまり、100%カバレッジを追求する事により、問題点の早期発見につながる事が分かります。なお、デザインの間違いを解決する方法は下記文献[2]を参照して下さい。

参考文献

- [1] SystemVerilog による検証の基礎、森北出版 2020. 第4章
[2] SystemVerilog によるモデリングと論理合成、共立出版 2026. 第6章

77 カバーグループ 2026.04.17

SystemVerilog ではファンクショナルカバレッジの機能を `covergroup` で指定しますが、`covergroup` のインスタンスを作らなければなりません。しかし、ベースクラスの `covergroup` をサブクラスで拡張する場合には、インスタンスを作る必要はありません。

ベースクラスにカバーグループが定義されていれば、ベースクラスのコンストラクタで `new` を使用してカバーグループのインスタンスが作られている筈です。実は、この `new` は、あたかも `virtual` メソッドのように動作します。つまり、サブクラスのコンストラクタから、`super.new` を呼び出すだけで、サブクラス用のカバーグループが作成されます。詳しくは、下記の記述例を見て下さい。

ベースクラス	サブクラス
<pre>class base_t; event ev; covergroup cg @ev; endgroup function new(); cg = new; endfunction endclass</pre>	<pre>class sub_t extends base_t; rand logic [2:0] port; constraint C { port inside {[0:5]}; } covergroup extends cg; coverpoint port { bins value[] = {[0:5]}; } endgroup function new(); super.new(); endfunction endclass</pre>
<p>ベースクラスには、空の <code>covergroup</code> が定義されていて、そのインスタンスが作られています。</p>	<p>サブクラスは、ベースクラスの <code>covergroup</code> を拡張して内容を追加していますが、そのインスタンスを作らずにカバレッジを行えます。<code>super.new</code> が呼ばれると、ベースクラスで作られる <code>cg</code> はサブクラス用の <code>covergroup</code> となります。</p>

78 type(expr) 2026.04.22

SystemVerilog の `type(this)` が便利な機能である事を紹介しましたが、その他の `type` 機能も便利です。

仮に、以下のような宣言があるとします。

```
int    i;
logic [3:0]  a, b;
```

`a+b` を計算した結果を格納するための領域を以下のように宣言できます。 `a` と `b` のタイプに依存せずに記述できる利点があります。例えば、`a` と `b` のタイプが `logic [3:0]` から `int` に変更されても以下の記述はコンパイルし直すだけで影響を受けません。

```
var type(a+b) c;
```

まあ、この記述法の利点はあまりないのですが、以下の記述は面白いです。

```
var type(i[24:20]) b5;
```

`b5` は 5 ビット符号なしです。確認をするためには、以下のようにすれば良いです。

```
$display("%s", $typename(b5));
```

`type` の機能を以下のように活用できます。こうすると、`a` と `va` の型は完全に一致します。

```
parameter type VAR_T = type(a);
VAR_T      va;
```

`type()` に指定する式は、データタイプも許されるので、以下のような指定ができます。

```
var type(bit signed[11:0]) bit12;
```

これらの機能を活用する事により、コードの保守は容易になります。ネットにも同様に適用できますが、`var` の代わりにネット型のキーワードを指定する必要があります。

79 RNG 2026.04.24

SystemVerilog にはミステリアスな概念・機能が多くありますが、RNG はその 1 つであると思えます。問題は、その仕様を正しく理解していなくても間違いを悟る事ができない点です。しかし、得られる結果が本来得られる結果と異なるのは確実です。

SystemVerilog の RNG の基礎を復習してみましょう。以下のような記述があるとします。a と b は全く同じ状況で乱数が割り当てられていますが、異なる乱数が発生されますか？

<pre>module test1; int a, b; initial begin: p1 a = \$random; end initial begin: p2 b = \$random; end endmodule</pre>	<ul style="list-style-type: none"> • initial プロシージャはシミュレータにより生成されるプロセスです。それぞれのプロセスは独自の RNG を保有します。 • シミュレータは乱数を発生する事によりプロセス用のシードを決定し RNG に割り当てます。 • したがって、左記の a と b には異なる乱数が発生されます。
--	--

以下の記述において、sample1.x と sample2.x には、異なる乱数が発生されますか？

<pre>class sample_t; rand int x; endclass module test2; sample_t sample1, sample2; initial begin: p3 sample1 = new; assert(sample1.randomize()); \$display("sample1.x = %0d", sample1.x); end initial begin: p4 sample2 = new; assert(sample2.randomize()); \$display("sample2.x = %0d", sample2.x); end endmodule</pre>	<ul style="list-style-type: none"> • クラスは、インスタンス毎に独自の RNG を保有します。 • new オペレータを起動しているプロセスは乱数を発生する事によりクラスインスタンスの RNG シードを決定します。 • したがって、sample1.x と sample2.x には異なる乱数が発生されます。
--	--

クラスのインスタンスは、どのプロセスから呼び出されるかわからないため、インスタンス毎に独自の RNG を保有している事に注意して下さい。

80 RNG 2026.04.25

SystemVerilog で乱数を発生するには、乱数発生機能を利用しますが、殆どの場合に小さな数が必要とするので、さらなる計算が必要になります。例えば、 $[0, 100)$ の乱数を得るには、`$urandom_range(0, 99)` または `$urandom%100` のように計算する場合があります。しかし、正確でしょうか？

検証においては、検証データを適切に発生させる事は優れたカバレッジに繋がります。したがって、乱数発生は極めて重要な要素となります。多くのテストケースを発生する前に、乱数発生法の質を確かめておくのは重要な作業の一部です。乱数の質を確認する方法の一つとして χ^2 テストが良く知られているので、利用するのも一つの方法です。発生法により、 χ^2 テスト結果が異なる事をご紹介します。

区間 $[0, r)$ にある数をランダムに発生させる試行を 1 万回繰り返して、 χ^2 テストで乱数発生機の質を確認します。公平を期するために、乱数発生機としては、Visual C++ の `rand()` を使用します。この関数の機能は SystemVerilog の `$urandom` とほぼ同じです。区間 $[0, r)$ の乱数を発生した場合、 χ^2 テスト結果が r に近ければ近い程良い乱数発生機であると判定されます。

以下に $r = 100$ のテスト結果をご紹介します。上段は乱数発生法を示し、下段は χ^2 テスト結果です。このテストでは、 χ^2 テスト結果が 100 に近ければ近い程、良い乱数発生機であると考えられます。

χ^2 テスト結果			
<code>rand()%100</code>	<code>(rand())>>1)%100</code>	<code>(rand())>>2)%100</code>	<code>(rand())>>3)%100</code>
104.28	91.18	99.18	92.68

r と繰り返し数に依存するので、一概に結論を出せませんが、得られた乱数から直接計算するのが一番良いとは限らない事が分かります。この実験の場合には、左から 3 番目の手法が一番良いと考えられます。一般的には、 χ^2 テスト結果が $[r - 2\sqrt{r}, r + 2\sqrt{r}]$ の範囲にあれば、良い乱数発生法と考えられます。

81 String 2026.05.05

SystemVerilog の string 機能は完備しているとは言えません。したがって、実用的な機能の開発はユーザに任されています。その際、string 処理機能はリエントラントな構造にしておく必要があります。同じ式内で何度も呼び出される可能性があるからです。

string 機能が完備していない事実は容易に理解できます。例えば、文字列の結合をする際、単純な結合しかできません。通常、結合する際には区切り記号などが必要になります。"OneTwoThree" ではなく、"One, Two, Three" のように表現したい場合が多くあります。文字列結合を以下のように表現できます。このファンクションはリエントラントです。

```
function automatic string concate(string s1,s2,sep);
    concate = {s1,s1.len?sep:"",s2};
endfunction
```

リエントラントなので、以下のように呼び出しのネストが可能になります。

```
$display("%s",concate(concate("Japan","Canada"," "),"France"," and "));
```

この場合には、以下のようにプリントされます。

```
Japan, Canada, and France
```

以下のような標準的な使用法もできます。

```
int    numbers[] = '{ 1, 2, 3, 4, 5 }';
string s;
s = "";
for( int i = 0; i < numbers.size(); i++ )
    s = concate(s,$sformatf("%0d",numbers[i]),", ");
$display("a: %s",s);
```

結果は、以下のようになります。

```
a: 1, 2, 3, 4, 5
```

concate のような機能は便利なので、SystemVerilog の学習時に作っておくと良いです。

82 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] 篠塚一也、SystemVerilog によるモデリングと論理合成、共立出版 2026.
- [9] SystemVerilog IEEE Std 1800-2023 の概要、アートグラフィックス 2024.