Artgraphics

SystemVerilog序説

初学者のためのオリエンテーション

篠塚一也

アートグラフィックス

Document Revision: 1.0,2024.09.23

www.artgraphics.co.jp

注意事項
（Caveat）

SystemVerilogの知識を個人的に習得する目的として本資料を活用して下さい。本資料を通して、業務（実践）で必要となるSystemVerilogに関する知識を習得して頂くのが本来の目的です。

転用目的（本来の目的と違った他の用途に使う事）で本資料を使用する事はご遠慮下さい。また、本資料から学んだ知識を転載する場合等は出典が本資料である事を明記して下さい。但し、他の著者の文書にも書かれている内容は、この限りではありません。本注意事項は現在及び過去に於ける弊社からの全てのフリーダウンロード資料に適用されます。

本注意事項に合意出来ない場合には、本資料を速やかに抹消して下さい。
尚、ダウンロード記録は、依然として残ります。

本資料の概要

- 本資料は、初学者を対象にしてSystemVerilogがどのような言語であるかを概説しています。決して易しく書かれているわけではありませんが、初学者が今後の方向を見定めるには十分な判断資料を提供しています。
- 本資料は、SystemVerilogが備えている機能の解説には深入りをせず、SystemVerilogが言語としてどのような様相と働きをする機能を備えているかを代表的な機能を通して解説しています。初学者にとっては理解し難い記述が多いと思いますが、“SystemVerilogが持つ雰囲気”を理解できるか否かが重要な点です。
- 雰囲気を理解できて、しかもSystemVerilogに興味を持てば、LRMまたは入門書を読む事により知識を習得できます。一方、“SystemVerilogは肌に合わない”と思える場合には、SystemVerilogの学習を止めた方が良いです。
- 本資料は、読み易さを考慮して設計分野と検証分野の二部に分けて解説しています。

第一部
設計のためのSystemVerilog

SystemVerilogは様々な機能を備えていますが、第一部では設計作業を支援するためのSystemVerilogの機能の概要を紹介します。それぞれの機能の解説は省かれていますが、どのような機能があるかという全体的な理解ができれば目標を達成できたと言えます。なお、packageとinterfaceは設計でも使用される重要な機能ですが、以下の概説では紹介を省きます。

モデリング

- SystemVerilogは、組み合わせ回路およびシーケンシャル回路をデザインするために使用されます。
- 回路種別によらず、alwaysプロシージャが使用されますが、組み合わせ回路とシーケンシャル回路では、センシティビティリストの表現が異なります。
- 組み合わせ回路のセンシティビティリストは、回路が依存する信号のリストです。組み合わせ回路の場合には、センシティビティリストを自動生成するalways_combが用意されています。
- シーケンシャル回路のセンシティビティリストとは、クロック信号とクロックに非同期な信号のリストで信号値の変化する状態も含みます。シーケンシャル回路に対しては、always_ffが用意されていますが、記述内容の妥当性を確認する目的であり、センシティビティリストの生成機能を持ちません。

組み合わせ回路の例

- 組み合わせ回路の例を以下に紹介します。デザインは、キーワードmoduleとキーワードendmoduleの間に記述します。下記のaluはモジュール名で、その後の()内にポートのリストが指定されています。
- alwaysの右にある@(s2,s1,s0,a,b)はセンシティビティリストですが、alwaysの代わりにalways_combを使用すれば、その指定を省略できます。

```
module alu(input enable,s2,s1,s0,[1:0] a,b,
           output logic [1:0] out);
logic [1:0]       _out;
assign out = enable ? _out : '0;
always @(s2,s1,s0,a,b)
           case ({s2,s1,s0})
           3'b001: _out = a&b;
           3'b010: _out = a|b;
           3'b100: _out = a^b;
           default _out = 'x;
           endcase
endmodule
```

この動作記述は、s2、s1、s0、a、bに依存するので、センシティビティリストは@(s2,s1,s0,a,b)となっています

シーケンシャル回路の例

- シーケンシャル回路の例を以下に紹介します。シーケンシャル回路のセンシティビティリストには、信号の直前にposedgeまたはnegedgeを指定しなければなりません。
- alwaysの内部記述は、非同期処理から始まり、同期処理がその後に続きます。内部記述ではクロック信号を使用してはいけません。

```
module data_shifter(input logic clk,reset,load,sl,sr,[3:0] data_in,
         output logic [3:0] data_out);
always @(posedge clk,posedge reset)
         if( reset )                                          ┐
                  data_out <= 0;                              ├ 非同期処理
         else if( load )                                      ┐
                  data_out <= data_in;                        │
         else if( sl )                                        ├ 同期処理
                  data_out <= {data_out[2:0],1'b0};           │
         else if( sr )                                        │
                  data_out <= {1'b0,data_out[3:1]};           ┘
endmodule
```

デザインの構成

- デザインは、以下のような構成になります（文献[3]）。

習得すべき知識

- 前ページで示した構造から、デザインをするためには以下のような知識が必要になります。
- ポート宣言の仕方
- デザインで使用する変数とネットの宣言法
- 他のデザインをインスタンスにする方法
- 連続代入文の記述法
- alwaysによる組み合わせ回路の記述法
- alwaysによるシーケンシャル回路の記述法

ポート宣言

- ポートの方向には、input、inout、output、refがあります。refポートは変数を直接引き渡します。
- 最初のポートに方向が付いていないと、inoutとなります。データタイプが省略されると、原則として、logicが採用されます。
- 2番目以降のポートに名称以外の指定が無い場合、その前のポートの定義が適用されます。名称以外の指定がある場合には、次の原則が適用されます。
- ポートの方向が省略されると直前のポートの方向を適用します。
- ポートタイプが省略されるとlogicになります。

ネットと変数の宣言

- ネットはネット型とデータタイプと次元を指定して宣言します。変数は、データタイプと次元を指定して宣言します。データタイプが省略されるとlogicが仮定されます。次元が省略されるとデータタイプが規定するビット数になります。

```
wire                a, b, c;
wire [1:0]          sum;
logic [3:0]         state;
var                 v;

assign sum = a+b;

// ...
```

宣言	オブジェクト
a, b, c	1ビットのlogic型ネット
sum	2ビットのlogic型ネット
state	4ビットlogic型変数
v	1ビットlogic型変数

デザインのインスタンス

- モジュール内には、他のモジュールのインスタンスを配置できます。
  SystemVerilogでは、配置する際に簡潔な接続法が準備されています。

```
universal_shift_register #(.NBITS(4)) DUT
          (.clk(clk),.reset(reset),.c1(c1),.c0(c0),
          .left_in(left_in),.right_in(right_in),
          .data_in(data_in),.q(q));
```

- ポートと接続する名称が一致する場合には、簡略記法 .* を使用できます。

```
universal_shift_register #(.NBITS(4)) DUT(.*);
```

- 一致しない名称があれば、簡略記法に追加する事ができます。

```
universal_shift_register #(.NBITS(4)) DUT(.*,.clk(clock));
```

連続代入文の記述

- 連続代入文はキーワードassignを使用して左辺（ネットまたは変数）と右辺を=オペレータで結びつけた代入文で、組みわせ回路を表現します。
- 右辺における信号または変数に変化が生じると右辺の式を評価します。その結果が、以前の値と異なれば左辺に設定されます。

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

右辺の式の評価結果が以前の値と異なる時のみ左辺が更新される

alwaysによる組み合わせ回路の記述法

- alwaysプロシージャを使用してRTL論理合成可能な組み合わせ回路を記述する際に守るべきルールは、以下のようになります。
- 組み合わせ回路が依存する全ての信号をセンシティビティリストに指定します。always_combやalways_latchを指定する場合には、この手順を省略できます。
- 動作をブロッキング代入文だけを用いて記述します。
- ラッチの生成を避けるためには、必ず、出力信号に値を設定しなければなりません。
- 動作記述にディレーやイベント制御を使用する事はできません。

組み合わせ回路の記述例

- 右の記述は正しい組み合わせ回路です。

```
module mux2(input a,b,sel,
            output logic out);
always @(a,b,sel)
            if( sel == 1'b1 )
                        out = a;
            else
                        out = b;
endmodule
```

- 右の記述は正しい組み合わせ回路記述ではないため、ラッチが生成されてしまいます.

```
module simple_latch(input [1:0] sel,
            output logic [3:0] out);
always @(sel)
            if( sel == 0 )
                        out = 0;
            else if( sel == 1 )
                        out = 4;
endmodule
```

alwaysによるシーケンシャル回路の記述法

- alwaysプロシージャを使用してRTL論理合成可能なシーケンシャル回路を記述するルールは、以下のようになります。
- クロック信号と非同期信号以外は、センシティビティリストに指定しません。
- 非同期信号を条件判定に使用する場合、条件判定とエッジが一致しなければなりません。
- クロック信号は、センシティビティリスト以外に指定する事はできません。
- 非同期信号は、必ず、条件判定に使用されなければなりません。
- 一般的に、ノンブロッキング命令（<=）を使用して記述します。

シーケンシャル回路の記述例

- 下記の記述は、正しいシーケンシャル回路の例です。

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

assign q_bar = ~q;

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

endmodule
```

クロック信号と非同期信号のみ指定する

negedge resetなので
if(reset == 0)
で条件判定をしている。

第二部
検証のためのSystemVerilog

SystemVerilogは、検証言語であると言われますが、既に紹介したようにSystemVerilogは設計機能も備えています。この第二部では、SystemVerilogの本質的な機能である検証機能を概説します。

検証機能の概要

- SystemVerilogの持つ検証機能は以下のように分類されます。
- 制約によるランダムスティミュラスの生成
- ファンクショナルカバレッジ
- アサーション
- 加えて、以下のような機能がありますが、これらの機能の紹介は省略します。
- クラス
- プログラム
- チェッカー
- クロッキングブロック
- インターフェース
- パッケージ
- プロセス生成
- プロセス間の通信機能
- 等等

制約による
ランダムスティミュラスの生成

SystemVerilogには、効率よく、かつ効果的にランダムスティムラスを生成する方法があります。制約を付けてランダムスティムラスを生成する事により、目的に即したテストデータを確実に準備する事ができます。現代の検証手法では、ランダムスティムラスを生成する方法はトランザクション生成の中心的な役割を担っています。

ランダム変数

- ランダム変数はrand、または、randcの修飾子を付けて宣言します。これらの修飾子をアレイ変数にも適用する事が出来ます。ランダム変数以外は、state変数と呼ばれます。
- ランダム変数はクラスのビルトインメソッドrandomize() により乱数が割り当てられます。
- ダイナミックアレイをランダム変数に指定すると、アレイ・サイズもランダムに決定されます。アレイ・サイズに制約を設定しないと膨大なアレイが生成されて、殆どシミュレーションが終了しない状況に陥ります。必ず、アレイ・サイズに制約を設定する事が必要です。

制約

- 単に乱数を発生するだけでは意味のあるテストデータを作成する事はできないので、制約を付ける事が必要になります。ただし、複雑過ぎる制約を与えると、制約ソルバーに負担がかかりシミュレーションのパフォーマンスが低下します。検証目的に即した適切な制約を設定する事が大切です。
- キーワードconstraintの後に制約名称を記し、{} の中に制約を列挙します。例えば、以下のように制約を定義すると、ランダム変数aとbは、a < b の関係が成立するように乱数が割り当てられます。

```
class sample_t;
rand logic [3:0] a, b;
constraint C { a < b; }
endclass
```

制約の種類

- 制約には以下のようなオペレータ、および文を使用する事ができます。
- insideオペレータ
- distオペレータ
- uniqueオペレータ
- implicationオペレータ（->）
- if-else制約
- foreach制約
- 乱数決定順序文（solve a before b）

ファンクショナルカバレッジ

近年の検証手法では、デザインの大規模化に対応するためのテスト法としてランダムスティミュラス生成機能を適用します。広範囲に及ぶ検証空間を効率良くカバーする方法としてランダムスティミュラス生成機能は重要な役割を果たすのは明らかです。大規模なデザインの検証を効率良く進めるためには、検証作業の生産性向上・自動化が必須な条件となります。そして、検証を自動化するためには、どのような値を持つデータが使用されて検証が行われたかを自動的に計測する仕組みが必要になります。

概要

- ファンクショナルカバレッジは仕様のどれだけの部分が検査されたかを示す指標です。デザインを検証する意味ではなく、寧ろ、検査者、又は、検査計画の進捗度を示します。ゴールは、勿論、100%のカバレッジです。
- ファンクショナルカバレッジは仕様を基にして確認をします。効率良く検証を遂行するためには、確認を漏れなく、しかも、重複が無いように進める必要があります。
- 例えば、以下の記述において、portには乱数が割り当てられます。検査でportが0から7の全ての値をとればカバレッジは100%となりますが、一部の値を取らない場合、乱数発生に制約を追加して取り得る値を制限しなければなりません。

```
rand bit [2:0]   port;
```

- したがって、乱数を発生した場合、実際に発生した値を計測する必要があります。ファンクショナルカバレッジはその計測機能を備えています。

カバレッジ仕様の定義

- covergroupはカバレッジ仕様を定義するための構文です。実際には、クラスと同じようにデータタイプです。covergroupには以下の機能を含む事が出来ます。
  - カバレッジ情報を取得するタイミングを示すクロックイベント
  - カバーポイント（coverpoint）
  - クロスカバレッジ（cross）
  - カバレッジオプション
- covergroupをクラスの中に定義する方法が最も一般的です。クラス内に定義したカバレッジ仕様は、embedded カバーグループと呼ばれます。

カバレッジ仕様の定義例

- 以下は、embeddedカバーグループの定義例です。

```
class sample_t;
rand bit [2:0]                  port;
rand bit [3:0]                  data;

constraint C_PORT { port inside {[0:5]}; };


covergroup cg;
          coverpoint            port { bins value[] = { [0:5] }; };
          portBydata: cross     port, data;
endgroup

function new();
          cg = new;
endfunction

endclass
```

仕様上、portは0~5の値しか取らない

cgというカバーグループにカバーポイントとクロスカバレッジを定義している

クロスカバレッジに名称を付けている

仕様上の制約に合わせてカバーポイントを定義する

クラス内にカバーグループを定義すると、その名称を持つ変数が自動的に確保されるので、変数をコンストラクタ内で初期化する

アサーション

アサーションは、システムの動作（即ち、仕様）を記述するための手段です。主として、アサーションはデザインの検証に使用されます。その他、ファンクショナルカバレッジ、または、DUTへの入力となるスティミュラスの検証にも使用されます。

概要

- アサーションではシーケンス、及びプロパーティを用いて仕様を記述して検証を行ないます。検証とは、仕様とデザインが一致する事を確認する作業です。アサーションには以下に示すような機能があり、仕様とデザインが一致するか否かをシミュレーションを通して確認します。
- assert：デザインが遂行しなければならない動作（仕様）を検証します。
- assume：シミュレータに対しては、成立しなければならない環境条件を検証する手段となります。フォーマルツールは、前提条件として使用します。
- cover：動作に対してのカバレッジを収集します。
- restrict：フォーマルツールへの制約を規定します。シミュレータには無効な命令です。

アサーションの種類

- アサーションには、即時実行型と並列型の二種類があります。
- 即時実行型アサーションは、通常の実行文と同じ様に動作し、即時に実行が終了します。
- 並列型アサーションは、通常のシミュレーションと並行して実行します。指定した検証条件が成立、または、不成立するまで検証が続行します。

即時実行型アサーション

- 即時実行型アサーションは、通常の実行文と同じ様に動作し、即時に実行が終了します。従って、時間を消費する概念がありません。次の例は、即時実行型アサーションを示しています。

```
packet_t  p;
...
p = new;
assert( p.randomize() )
        else $fatal(0,"packet_t::randomize() failed.");
...
```

- 即時実行型アサーションは時間間隔の概念を持たないので、現在の状態を確認する目的で使用されます。

並列型アサーション

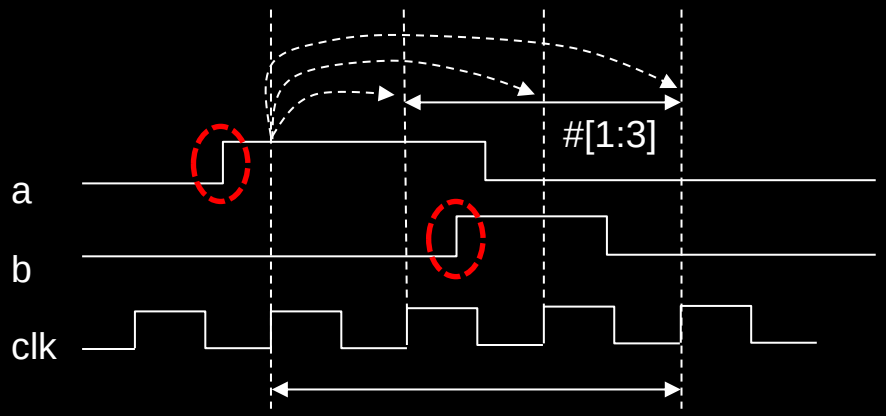
- 並列型アサーションはアサーションの仕様に基づき、デザインのシミュレーション結果と仕様を照合しながら実行して行きます。一致しない点があれば、検証は失敗し、その時点で終了します。仕様の最後まで一致すれば検証は成功と判定されます。
- 下記の記述は並列型アサーションの一例です。

```
module test(input clk,a,b);

          default clocking cb @(posedge clk); endclocking

          property check_a_b;
                    $rose(a) |-> ##[1:3] $rose(b);
          endproperty

          assert property (check_a_b)
                    else $display("@%0t: FAIL",$time);
// ...
endmodule
```

並列型アサーション

- プロパーティcheck_a_bは以下のような仕様を表現しています。

```
property check_a_b;
         $rose(a) |-> ##[1:3] $rose(b);
endproperty
```



この区間で $rose(b)==1'b1になれば仕様を満たします。

あとがき

まとめ

- 本資料では、設計機能と検証機能に分けてSystemVerilogの全体像を紹介しました。初学者にとって、内容は決して易しくないので、何度か読み直してSystemVerilogの概要を把握して下さい。そして、今後は進路にしたがって、必要な知識を習得して下さい。
- 本資料の説明に含まれている用語の理解を深めていくのも一つの進め方です。或は、入門書を読み始めるのも進め方の一つです。何れにせよ、SystemVerilogに興味を持つ方は、学習計画を立てて進めると良いです。
- SystemVerilogは肌に合わないという方には、SystemVerilogの学習をすすめません。時間を少しおいて再度読み直すか、或は他の選択肢を検討するのも大切です。

参考文献

ここで紹介したSystemVerilog機能は下記の文献を学習すれば理解できます。

文献[1]は、LRMと呼ばれるSystemVerilogのリファレンスマニュアルです。丹念に読めば、SystemVerilogを完全に理解できます。

文献[2]はSystemVerilog全体の知識を習得するために適しています。実践で必要となる知識が徹底的に、かつ完全に解説されています。

文献[3]は、設計分野で必要とされるSystemVerilogの知識が非常に詳しく解説されています。特に、オペレータに関する詳しい記述が含まれています。初学者が、最初に読むべき入門書です。

文献[4]は、SystemVerilogの検証機能を詳しく解説した参考書です。UVMに関する解説も含まれています。

[1] IEEE Std 1800-2023: IEEE Standard for SystemVerilog – Unified Hardware Design, Specification and Verification Language.
[2] 篠塚一也、SystemVerilog入門, 共立出版 2020.
[3] 篠塚一也、SystemVerilog超入門, 共立出版 2023.
[4] 篠塚一也、SystemVerilogによる検証の基礎、森北出版 2020.