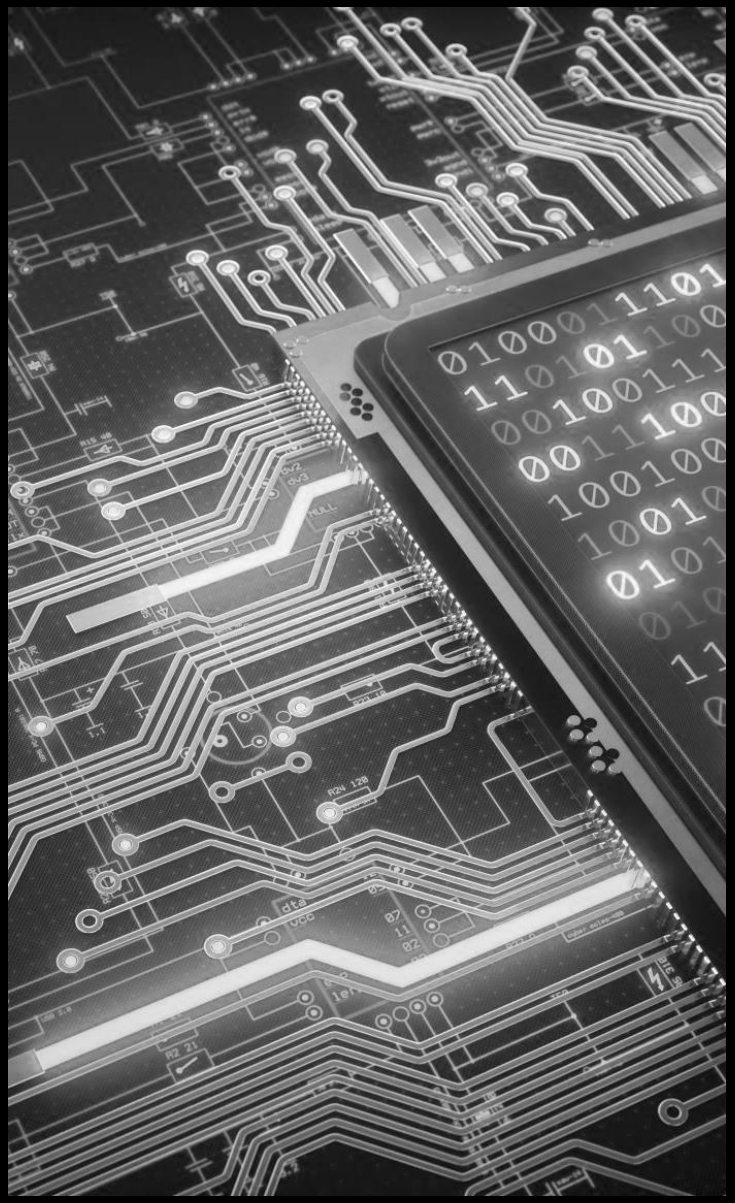
Artgraphics



SCL
SystemVerilogクラスライブラリー

生産性向上のためのSystemVerilogパッケージ

はじめに

- このスライドでは、他のSystemVerilogクラスライブラリーとの差異を中心に解説します。したがって、検証環境構築で必要とされる標準的な機能の解説は簡単な紹介にとどめます。

SCLとは

- SCLは、検証環境を構築するためのSystemVerilogクラスライブラリーです。
- SCLは、TLM（Transaction Level Modeling）を採用し、トランザクションを使用してシミュレーションしDUTの検証を行います。
- SCLには、検証コンポーネントを定義するためのベースクラス、トランザクションを定義するためのベースクラス、およびトランザクション生成手順を定義するためのベースクラス等が用意されています。
- SCLは、パターンマッチング機能、文字列操作機能、プリント書式機能、クロック生成機能、プロセス生成機能、プライオリティキュー、ファイル入出力等の汎用的な機能も同時に備えているので、検証環境構築作業の省力化を実現します。
- SCLは、新しいアーキテクチャを採用する事により軽量であるため、コンパイルが早く、しかも、生成される実行モジュールが小さいという利点があります。
- SCLの一部の機能はUVMと一緒に使用する事もできます。

SCLの特長

- virtualインターフェースの操作が簡単です。マクロを使用して、virtualインターフェースを操作するためのクラスを定義すると、set() でvirtualインターフェースの設定、get() でvirtualインターフェースの取得をできます。
- TLMポートを備えたコレクターが標準的に装備されています。モニターにもTLMポートが標準的に装備されています。
- スコアボードは、DUTの出力が正しいか否かを検証するためのコンポーネントとして必要な機能を備えています。TLMポートは言うまでもなく、predict()メソッドも備えているので、スコアボードは期待される結果とDUTからの出力との照合を行える状態になっています。
- SCLではトランザクションの生成手順をツリーで表現するので、生成手順は明快です。
- SCLは強力なプリント機能を備えているので、殆どのプリント処理は省力化されます。ビット幅や表現形式を変更してもSCLが自動的にカラム位置を調節します。
- 強力で汎用的な機能が備えられているので、DPI-Cへの依存度が減少します。例えば、SCLはパターンマッチング機能を備えているのでDPI-Cを使用する必要はありません。

検証環境構築のためのベースクラス

- 以下のようなベースクラスが用意されています。

SCLクラス	機能
scl_driver_t	ドライバーのベースクラスです。ユーザはドライバーをこのクラスのサブクラスとして定義しなければなりません。ドライバーは、ジェネレータからトランザクションを取得する機能を備えています。
scl_generator_t	ジェネレータのベースクラスです。ジェネレータは、ドライバーの要求に応じてトランザクションを生成して戻します。実際には、シナリオにより割り当てられたプレイを介して、トランザクションを生成して貰います。
scl_collector_t	コレクターのベースクラスです。コレクターは、DUTからのレスポンスをサンプリングしてトランザクションに変換して、モニターにトランザクションを送信する役目を持ちます。一般的には、コレクターは検証に関わる作業を担当しません。
scl_monitor_t	モニターのベースクラスです。モニターは、コレクターから受信したトランザクションを他の検証コンポーネントに一斉に転送します。モニター自身も簡単な検証作業を行います。
scl_agent_t	エージェントのベースクラスです。エージェントは、ドライバー、ジェネレータ、コレクター、モニターから構成される最小単位の階層的検証コンポーネントです。
scl_env_t	エンバイロンメントのベースクラスです。エンバイロンメントは、エージェントや他のエンバイロンメントから構成される階層的な検証コンポーネントで、検証内容に応じて様々な規模のエンバイロンメントが開発されます
scl_scoreboard_t	スコアボードのベースクラスです。スコアボードは、DUTからのレスポンスを検証する役目を持ちます。
scl_test_t	テストのベースクラスです。テストは、一般に、複数のエンバイロンメントから構成されますが、テスト同士が共有する資源をベースクラスとして定義するのが一般的です。

ドライバーの定義例

```
class driver_t extends scl_driver_t#(simple_item_t);
vif_config::vif_type    vif;
`scl_component_m(driver_t)
`scl_component_new_m
`scl_extern_connect_phase_m
`scl_extern_run_phase_m
extern task drive_dut(TR item);
endclass

`scl_connect_phase_m(driver_t)
        super.connect_phase(param);
        vif = vif_config::get();
`scl_end_connect_phase_m

`scl_run_phase_m(driver_t)
TR      item;
        m_get_port.m_connected_port.set_option(SCL_PORT_REUSE_TRANSACTION);
        forever begin
                m_get_port.get(item);
                drive_dut(item);
                @(negedge vif.clk);
        end
`scl_end_run_phase_m

task driver_t::drive_dut(TR item);
        vif.reset <= item.reset;
        vif.load <= item.load;
        vif.up_down <= item.up_down;
        vif.d <= item.d;
        if( item.reset )
                vif.reset = #1 0;
endtask
```

検証コンポーネントのインスタンス

- 検証コンポーネントのベースクラスを使用して定義したコンポーネントに対して、以下のようにインスタンスを作ります。

```
monitor = `scl_create_component_m(monitor_t,"monitor",this);
```

- このようにモニターのインスタンスを作っておくと、ソースコードを修正せずに、monitor_t と互換性のあるモニターで置き換えられます。
- 例えば、カバレッジ計算を行うモニターのサブクラス monitor_fc_t を作ったとすると、以下の命令で monitor_t が monitor_fc_t に置き換わります。

```
`scl_change_type_m(monitor_t,monitor_fc_t)
```

トランザクション操作のためのベースクラス

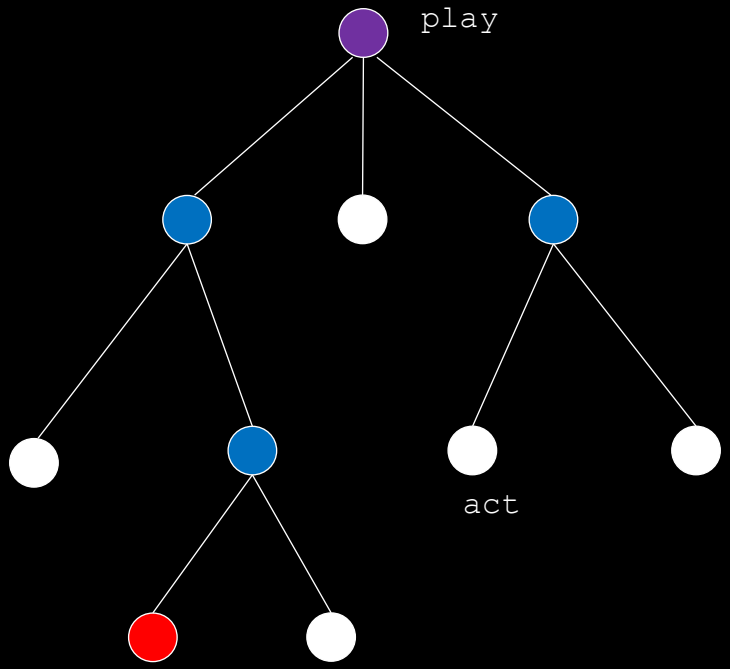
SCLクラス	機能
scl_scenario_t	シナリオのベースクラスです。ジェネレータとプレイを結びつける役目を持ちます。
scl_play_t	トランザクションを作る手順のベースクラスです。トランザクション生成手順はツリーで表現されますが、このクラスはツリーのルートを表現するために使用されます。
scl_act_t	トランザクションを作るためのベースクラスです。ツリーの内部ノードおよびリーフノードを表現するために使用されます。リーフノードが一つのトランザクションを生成します。内部ノードは、トランザクションを生成するためのマクロ的なスクリプトの役目を持ちます。
scl_transaction_t	トランザクションを定義するためのベースクラスです。

トランザクション生成手順

- トランザクション生成手順は、シナリオ、プレイ、アクトより組み立てられます。
- シナリオは、ジェネレータにプレイを割り当てます。
- プレイは、生成手順を表現するためのツリーのルートノードとして割り当てられて、ツリーを走査する制御を行います。
- アクトは、ツリーの内部およびリーフノードを表現します。内部ノードにはループ処理を記述できるので複雑なツリーを構築できます。
- リーフノードが一つのトランザクションを生成します。
- リーフノードをボトムアップでサブツリーを構成する事により、複雑なトランザクション生成手順を定義できます。したがって、トランザクション生成処理を簡単に部品化（ライブラリー化）できます。
- ルートノードは、ツリーのリーフノードを左から右へ順に走査して一連のトランザクションを生成します。
- ドライバーがジェネレータにトランザクションを要求する度に、リーフノードが起動されます。

トランザクション生成手順例

- ユーザは、トランザクションを生成する手順をツリー状に定義するだけで良いです。



トランザクション生成手順のベースクラス

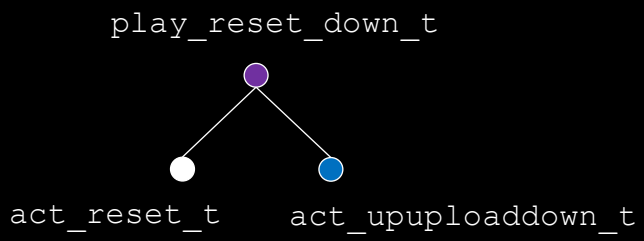
- テストケースではトランザクション生成手順を共有する事がしばしば発生するので、共有を可能にするためにプレイのベースクラスを定義しておくと効率が良くなります。

```
class play_t extends scl_play_t#(simple_item_t);
act_reset_t          act_reset;
act_down_t           act_down;
act_up_t             act_up;
act_load_t           act_load;
act_upuploaddown_t   act_upuploaddown;
`scl_object_m(play_t)
`scl_object_new_m
endclass
```

プレイによるテストケースの作成

- プレイのベースクラスを使用してテストケースを準備できます。

```
class play_reset_down_t extends play_t;
`scl_object_m(play_reset_down_t)
`scl_object_new_default_m("play_reset_down");

task get_item();
          `scl_act_get_item_m(act_reset,this)
          `scl_act_get_group_m(act_upuploaddown,this)
endtask

endclass
```



リーフノードの定義

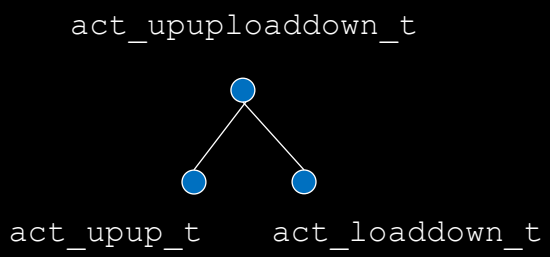
- リーフノードを表現するアクトを以下のように定義できます。

```
class act_reset_t extends scl_act_t#(simple_item_t);
`scl_object_m(act_reset_t)
`scl_object_new_m
extern function void make_item();
endclass

function void act_reset_t::make_item();
          m_play.m_item.reset = 1;
          m_play.m_item.d = 0;
          m_play.m_item.up_down = 0;
          m_play.m_item.load = 0;
endfunction
```

ツリー内部ノードの定義

- 内部ノードを以下のように定義できます。

```
class act_upuploaddown_t extends scl_act_t#(simple_item_t);
act_upup_t          act_upup;
act_loaddown_t      act_loaddown;
`scl_object_m(act_upuploaddown_t)
`scl_object_new_m

task get_group();
          `scl_act_get_group_m(act_upup,m_play);
          `scl_act_get_group_m(act_loaddown,m_play);
endtask

endclass
```

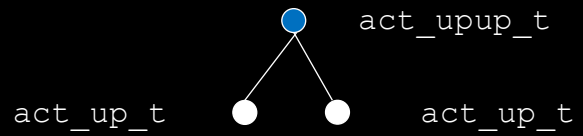


ツリー内部ノードの定義の特徴的機能

- 内部ノードの記述では、繰り返しやif文等の条件判定文を自由に使用できるので汎用的な生成手順を定義できます。

```
class act_upup_t extends scl_act_t#(simple_item_t);
act_up_t    act_up;
`scl_object_m(act_upup_t)
`scl_object_new_m

task get_group();
    `scl_act_get_item_m(act_up,m_play)
    `scl_act_get_item_m(act_up,m_play)
endtask

endclass
```

```
class act_upup_t extends scl_act_t#(simple_item_t);
act_up_t    act_up;
`scl_object_m(act_upup_t)
`scl_object_new_m

task get_group();
    repeat( 2 )
        `scl_act_get_item_m(act_up,m_play)
endtask

endclass
```



シナリオによるプレイの決定

- テスト用の検証コンポーネントにおいて、シナリオを起動してプレイを割り当てます。

```
class test1_t extends test_base_t;
`scl_component_m(test1_t)

function new(string name="test1",scl_component_t parent=null);
         super.new(name,parent);
endfunction

function void build_phase(scl_run_param_t param);
         super.build_phase(param);
         scl_scenario_t::allocate("test1*generator",
                  `scl_create_object_m(play_reset_down_t,"play"));
endfunction
endclass
```

virtualインターフェース

- マクロを使用してvirtualインターフェース用のクラスを作り、set()とget()でvirtualインターフェースの設定と取得を行います。

```
interface simple_if(input logic clk);
...
endinterface
`scl_vif_config_m(virtual simple_if,vif_config)
```

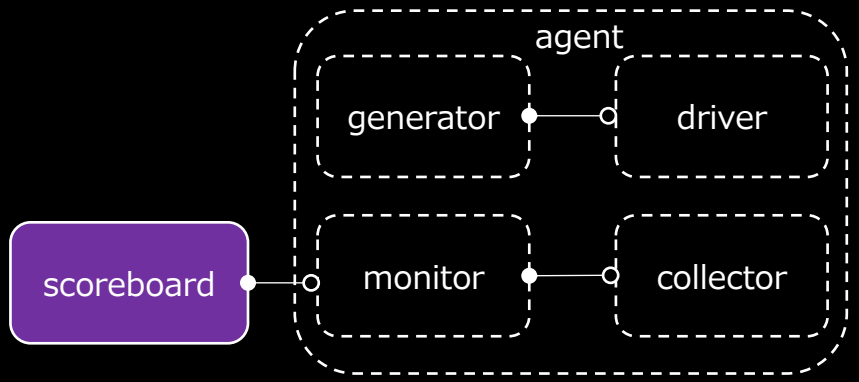
```
module top;
bit       clk;

simple_if SIF(.*);
device DUT(SIF);
...
initial begin
          vif_config::set(SIF);
          ...
end
endmodule
```

```
class driver_t ...;
vif_config::vif_type          vif;
...
vif = vif_config::get();
...
endclass
```

スコアボード

- スコアボードにはTLMポートが標準装備されているので、他の検証コンポーネント（例えば、モニター）からトランザクションを受信できます。
- 受信したトランザクションを基にして、predict()メソッドにより計算結果を予測し、トランザクションに記録されているDUTからの出力と照合して検証します。

```
function void scoreboard_t::verify(TR item);
scl_print_data_s        row[$], column;
int                     judge;
expected_item.copy(item);
predict(expected_item);
judge = expected_item.q == item.q;
...
endfunction
```



検証コンポーネントの階層走査

- 階層を走査する機能も装備されています。例えば、部分階層のルートを指定するだけで階層順にインスタンスを取り出せます。

```
scl_instance_s          q[$];
string                  name, tabs;
scl_get_instances_by_preorder(root,q);
foreach(q[i]) begin
            tabs = {q[i].m_depth{"  "}};
            scl_system($sformatf("%s%s",
              tabs,q[i].m_component.get_full_name()));
end
```

```
test
  test.env0
    test.env0.agent0
      test.env0.agent0.collector0
      test.env0.agent0.driver0
      test.env0.agent0.generator0
      test.env0.agent0.monitor0
  test.env1
    test.env1.agent1
      test.env1.agent1.collector1
      test.env1.agent1.driver1
      test.env1.agent1.generator1
      test.env1.agent1.monitor1
```

検証環境の定義状態

- 定義されている全てのオブジェクトクラスを簡単に取り出せます。例えば、以下のようにすると、全てのオブジェクトクラス名を取り出せます。

```
string    name[$];
scl_get_object_types(name);
```

- 定義されている全ての検証コンポーネントクラスは以下のように取り出せます。

```
string    name[$];
scl_get_component_types(name);
```

エレガントな診断機能

- クラスに定義されているプロパーティは、print()メソッドによりエレガントにプリントされます。なお、print()メソッドはクラス定義内容の診断機能であり、検証結果を確認するプリント機能ではありません。
- 下記の例では区切り記号を挿入していますが、そのオプションを外すこともできます。

```
================================================================
Name              Type              Size     Value
----------------------------------------------------------------
simple_item       simple_item_t     -        @1
  a               integral          64       'hxxxx_xxxx_abcd_ef01
  sum             int               32       1,234,567,890
================================================================
```

強力なプリント書式機能

- レポートのヘッダ情報を定義するデータ構造を使用すると、汎用的なプリント機能を開発できます。
- カラム名称、ビット幅、表現形式（2進、16進等）等の変更が発生してもプリント処理を変更する必要はありません。
- データ構造の定義情報と使用するマクロを変更するだけでSCLが自動的にカラム位置を調節します。

強力なプリント書式機能例

- 以下に簡単な例を紹介します。ビット幅の変更、表現形式の変更、カラムの入れ替えをしてもカラム位置は自動的に調節されます。

```
=========================
time  reset d      q
-------------------------
@ 10: 0     'hc778 'hc778
@ 30: 0     'hc8d0 'hc8d0
@ 50: 0     'h1471 'h1471
@ 70: 0     'h2004 'h2004
@ 90: 0     'hea38 'hea38
@100: 1     'h17ef 'h0000
@110: 0     'h17ef 'h17ef
@130: 0     'h8562 'h8562
@150: 0     'hfc26 'hfc26
@170: 0     'h3b02 'h3b02
@190: 0     'h02a2 'h02a2
=========================
```

WIDTH==16

```
===================================
time  reset d           q
-----------------------------------
@ 10:     0 'he093_c778 'he093_c778
@ 30:     0 'h060c_c8d0 'h060c_c8d0
@ 50:     0 'h3154_1471 'h3154_1471
@ 70:     0 'h870a_2004 'h870a_2004
@ 90:     0 'ha9bb_ea38 'ha9bb_ea38
@100:     1 'h700e_17ef 'h0000_0000
@110:     0 'h700e_17ef 'h700e_17ef
@130:     0 'had45_8562 'had45_8562
@150:     0 'haf37_fc26 'haf37_fc26
@170:     0 'h848b_3b02 'h848b_3b02
@190:     0 'hd0cf_02a2 'hd0cf_02a2
===================================
```

WIDTH==32

```
=========================
time  reset q      d
-------------------------
@ 10: 0     'hc778 'hc778
@ 30: 0     'hc8d0 'hc8d0
@ 50: 0     'h1471 'h1471
@ 70: 0     'h2004 'h2004
@ 90: 0     'hea38 'hea38
@100: 1     'h0000 'h17ef
@110: 0     'h17ef 'h17ef
@130: 0     'h8562 'h8562
@150: 0     'hfc26 'hfc26
@170: 0     'h3b02 'h3b02
@190: 0     'h02a2 'h02a2
=========================
```

WIDTH==16でdとqのカラムを入れ替えた例

豊富なユーザのためのマクロ

- SCLの特長の一つは、ユーザのための強力なマクロが豊富な点です。
- 例えば、マクロを使用するとジェネレータの記述を下記の一行で済ませる事ができます。

```
`scl_generator_m(generator_t,simple_item_t)
```

- 標準的なエージェントであれば、以下のマクロで済ませられます。

```
`scl_agent_m(agent_t,,_t)
```

豊富な文字列処理機能

- SCLには、検証作業で必要となる文字列処理機能が網羅されています。

scl_max_len	scl_reverse_string	scl_find_last_substr	scl_separate_string
scl_is_alnum	scl_starts_with	scl_trim	scl_strncmp
scl_is_alpha	scl_ends_with	scl_replace_head_ws	scl_strnicmp
scl_is_digit	scl_first_index	scl_replace_tail_ws	scl_strnset
scl_is_lower	scl_first_index_from	scl_replace_first	scl_ascii_to_hex
scl_is_upper	scl_last_index	scl_replace_last	scl_ascii_to_bin
scl_tolower	scl_last_index_from	scl_replace_all	scl_ascii_to_oct
scl_toupper	scl_find_first_substr	scl_replace_substr	

豊富な文字列処理機能例

- 例えば、部分文字列の比較を以下のように簡単に行えます。

```
if( scl_ends_with(pathname,".driver") )
          ...
```

強力なパターンマッチング機能

- SCLには、パターンマッチング機能が備わっているのでDPI-Cに依存する必要はありません。

パターン	match	unmatch
abc	abc	ab, abcd
a*	a, ab, abc	b
a*t	abt, abcdeft, abtt, at	ab
a?xyz	abxyz	axyz
[agw]*[ne]	apple, grape, watermelon	america
[a-g]*	apple, banana, grape	watermelon
e	apple, eagle, meta	book
*test¥[[0-9]¥]	test[1], test[5], top.test[2]	test[10]

強力なパターンマッチング機能例

- パターンマッチングを容易に記述できます。

```
string    pattern = "test[1-5]*.driver[0-3]";
...
if( scl_pattern_matching(pattern,"test1.env0.agent0.driver0") )
          ...
```

豊富なプリント書式マクロ

- 複雑な形式のプリントも書式マクロを使用すると簡単に記述できます。

`scl_sformatfb_m(v)	`scl_sformatfs_m(v)	`scl_breakupd_m(v)	`scl_nsprintfb_m(v)
`scl_sformatfo_m(v)	`scl_breakupb_m(v)	`scl_sprintfb_m(v)	`scl_nsprintfo_m(v)
`scl_sformatfh_m(v)	`scl_breakupo_m(v)	`scl_sprintfo_m(v)	`scl_nsprintfh_m(v)
`scl_sformatfd_m(v)	`scl_breakuph_m(v)	`scl_sprintfh_m(v)	`scl_nsprintfd_m(v)

豊富なプリント書式マクロ例

- 例えば、以下のように使用できます。

使用例	実行例
`$display("%s = %s",`scl_nsprintfb_m(a));`	a = 32'b0000_0001_0010_0011_1010_1011_1100_1101
`$display("%s = %s",`scl_nsprintfh_m(a));`	a = 32'h0123_abcd
`$display("%s = %s",`scl_nsprintfd_m(a));`	a = 123,456,789
`$display(“%s = %s”,`scl_name_m(a),`scl_sformatfb_m(a));`	a = 00000001001000111010101111001101

その他の機能

- SCLには、その他にも便利な機能が満載されています。
- メッセージプリント機能
- クロック生成機能
- プロセス生成機能
- ファイル入出力機能
- プライオリティキュー
- パラレルアレイ
- ユーティリティ

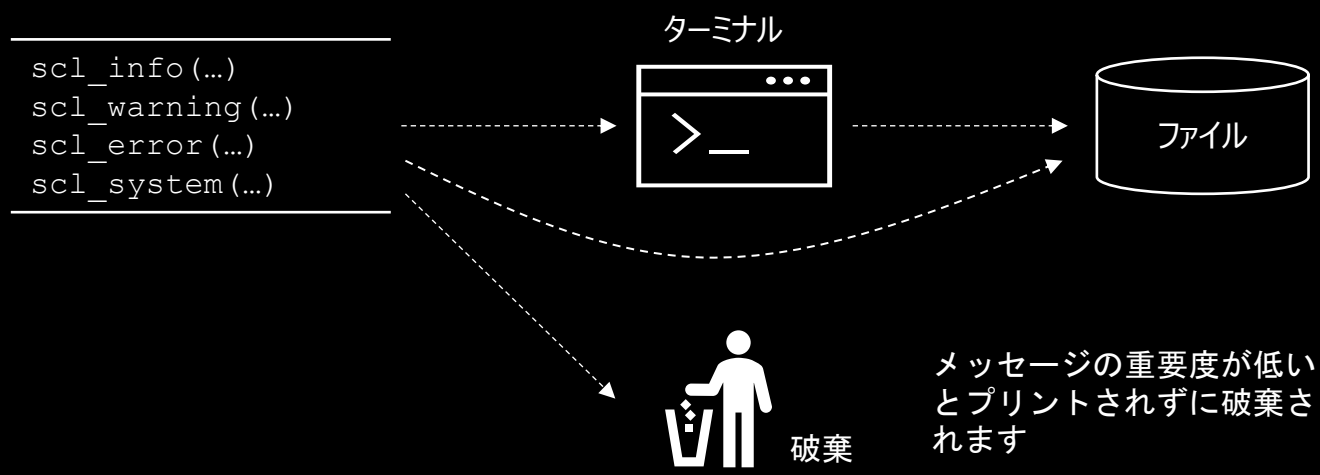
メッセージプリント機能

- SCLのプリント機能（scl_info()、scl_warning()、scl_error()、scl_system()）を使用すると、ターミナル出力をファイル出力に簡単に切り替える事ができます。
- この機能により、検証結果をユーザ指定のファイルに安全に保存できます。なお、$displayや$writeによるプリント情報は対象外となるので、ユーザは柔軟性のあるプリント処理を実現できます。
- SCLのプリント機能でプリントするとメッセージレベルも同時にプリントされるので、プリントされた情報が識別し易いという利点があります。状況に合わせてプリント機能を使い分けられます。

```
SCL_INFO: Simulation started at test1_t
SCL_ERROR: Test must be provided in command line as '+scl_testname=test_name'
```

メッセージプリント機能

- プリントの出力先を簡単に切り替える事ができます。
- ターミナルとファイルの両方に出力する事もできます。



メッセージプリント機能例

- テストケース単位に検証結果をファイルに記録できます。
- 以下のようにプリントファイルを設定し、SCLのプリント機能でプリントするとメッセージはターミナルの代わりにファイルに書き込まれます。勿論、ターミナルへの出力も継続できます。

```
string    log_filename = "test_read_modify_write";
...
if( !scl_set_print_file(log_filename) )
          scl_error($sformatf("Couldn't make log-
file: %s",log_filename));
...
```

ファイル入出力機能

- SystemVerilogの機能を最大限に活用したファイル入出力機能が備えられています。
- 例えば、検証結果をキューに保存しておき、シミュレーション終了後にまとめてファイルに出力できます。並列処理では、複数のプロセスがメッセージを同時にプリントするため、メッセージが混ざり合い見難くなります。そのため、検証結果をまとめて出力する必要があります。このような場合には非常に便利です。
- あるいは、ファイルの内容をキューに読み込む事もできます。この機能により、ファイルの内容を再構成したり、追加処理ができます。ログファイルの管理に便利な機能です。

ファイル入力例

- 下記の例は、ファイルに準備されているテストデータをアレイに読み込んでいます。

```
module test;
import scl_pkg::*;
string    filename = "SystemData/tmp_file.dat",
          lines[];
int       code;
initial begin
          code = scl_get_lines(filename,lines);
          if( code ) begin
                    foreach(lines[i])
                              $write("%s",lines[i]);
                    $display("%0d lines read",lines.size);
          end
end
endmodule
```

ファイル出力例

- 下記の例は、ランダムに文字列を生成して、テスト用のデータをファイルに準備しています。

```
module test;
import scl_pkg::*;
string    lines[5],
          filename = "SystemData/tmp_file.dat";

initial begin
          foreach(lines[i])
                    lines[i] = scl_random_string(SCL_ALPHA,10,20);
          if( scl_put_lines(filename,lines) )
                    $display("%s was written.",filename);
end
endmodule
```

プライオリティキューの応用例

- プライオリティキューは多くの分野で使用されます。例えば、以下のような状況に適用できます。

メールボックスにメールが不定期な間隔で届くとします。ここで、メールは0と1から構成されるシーケンス（例えば、"010111"）です。同時に、他の並列プロセスにより不定期の時間間隔でメールは取り出されるとします。そして、メールボックスからメールを取り出す時には、最大長を持つメールを取り出さなければならないとします。

- この問題を解くには、最大長を持つメールが常に認識されている必要があります。メールを書き込む時、あるいは、メールを取り出す時にメールをメッセージ長でソートする事も考えられますが、メールの書き込み、または、取得時に毎回ソートするのは効率が良くないので実用的ではありません。このような状況では、プライオリティキューの使用が最適です。プライオリティキューは、全体をソートせずに最大値、または、最長値を効率よく取り出せる機能を持ちます。

ユーティリティの例

- 現在時刻を簡単に取り出せます。

```
scl_info($sformatf("Simulation started at %s",scl_date()));
```

- 結果は以下のようにプリントされます。

```
SCL_INFO: Simulation started at Sat Feb  3 13:14:37 JST 2024
```

まとめ

- 以上、SCLの特徴的な機能を中心に解説しましたが、検証環境構築で使用される標準的な機能に対しても、SCL独自の手法が活かされて使い易い機能を実現しています。SCLは、新しい考えを基に開発されたSystemVerilogクラスライブラリです。
- SCLをUVMと共に使用できるので、UVMのメソドロジークラスで検証コンポーネントを記述し、シミュレーションして、検証結果のレポートをSCLで汎用的に表現すると検証作業の生産性と効率が向上します。
- 本製品に関するお問い合わせは、下記の連絡先をご利用下さい。
  連絡先：contact.us@artgraphics.co.jp