JEVeC DAY 2018 チュートリアル資料

Artgraphics

SystemVerilog from Zero to One

（検証の基礎からUVM適用までの実践知識を習得）

【ダイジェスト版】

篠塚一也

アートグラフィックス

2018.12.11

www.artgraphics.co.jp

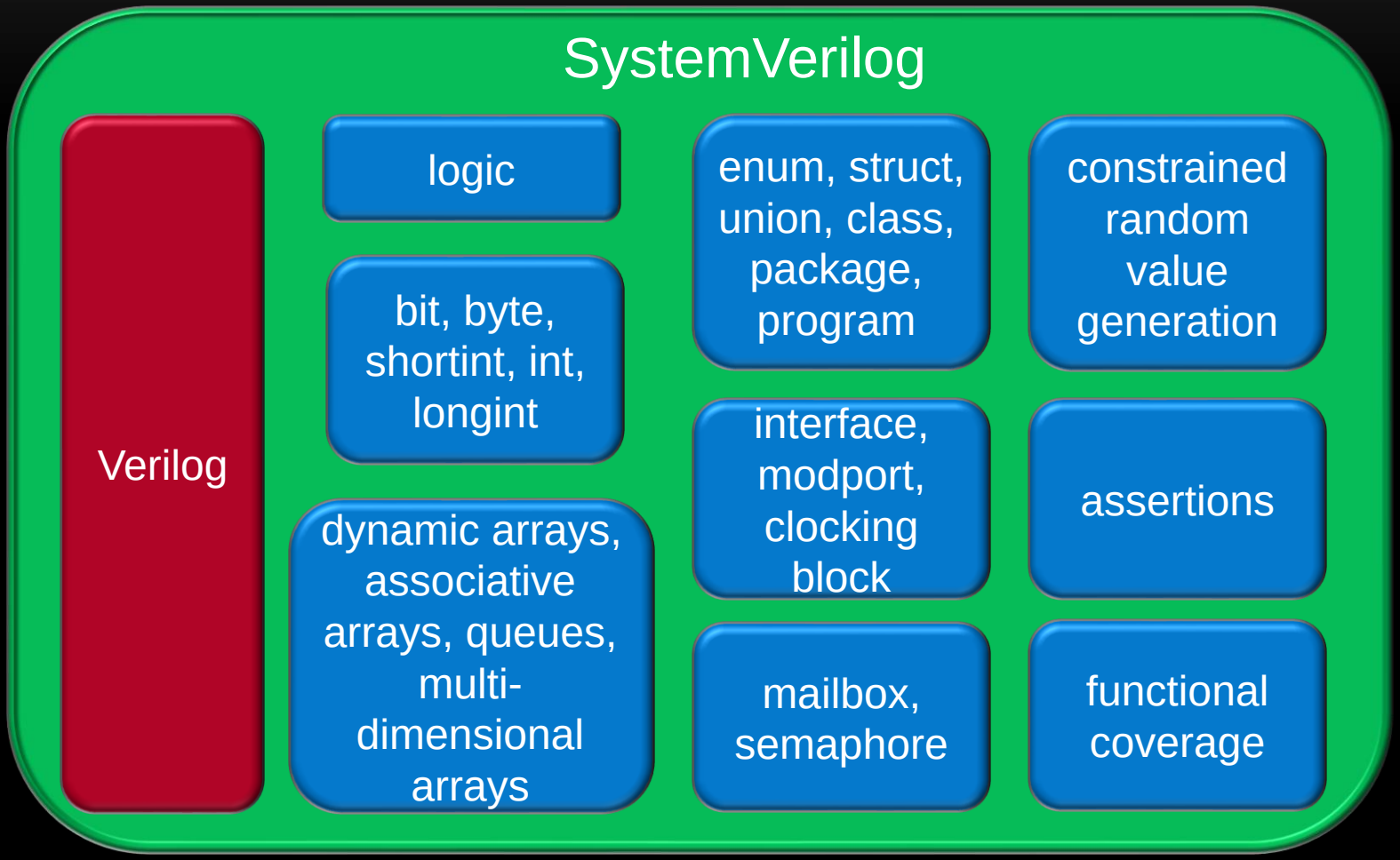
本チュートリアルの概要

- 本チュートリアルで使用する資料はダイジェスト版です。フル・テキスト版はJEVeCのホームページからダウンロードして下さい。
- 本チュートリアルでは、SystemVerilogによる検証の基礎知識を総括します。その上で、検証作業に要求される実践知識をUVMを例にとり解説します。
- SystemVerilogが持つ検証機能としては、ファンクショナル・カバレッジ、アサーション、ランダム・スティミュラスの生成機能等を概説します。
- このチュートリアルは、SystemVerilogの最新仕様（IEEE Std 1800-2017）、及び、UVM 1.2 をベースにしています。

SystemVerilogの特徴

SystemVerilogが備える特徴的な機能を紹介しVerilog HDLとの差異を明確にします。その差異を通じてSystemVerilogが言語として目指す役割を解説します。

SystemVerilogはVerilogのsuperset



SystemVerilogの代表的な機能を羅列しました。この他にも、SystemVerilogには重要な機能があります。

Verilog HDLの問題点（変数の初期化順序）

- Verilog HDLにはタイミングに関して曖昧な機能が多々あり、SystemVerilogはそれらを解決しています。
- 例えば、Verilog HDLでは、宣言時に指定した変数の初期化（inline initialization）はシミュレーションの開始と同時に行われます。時刻0で変数の初期化処理を記述すると、予測しない状況が発生します。
- 下記の例に於いて、Verilog HDLでは6行目の文が2行目の文の後に実行される保証はありません（non-deterministic）。

```
1  module test;
2  integer width = 5;
3  integer size;
4
5          initial begin
6                  size = width;
7          end
8  // ...
9  endmodule
```

- SystemVerilogでは、変数の宣言時に指定した初期化処理はシミュレーション開始前に行われる為、上記の例は正しく動作します。即ち、6行目の文が実行する前に、2行目の文が実行します。

Verilog HDLの問題点
(変数の初期化は時刻0でイベントを誘発)

- Verilog HDLでは、イベント待ちの変数にinline initializationが指定されていると、正しい結果を生成するとは限りません。
- Verilog HDLでは、3行目の初期化処理と10行目のイベント待ちが同時に実行する為、実行順序により結果が異なります。実行する都度、異なる結果が得られる可能性が高いです。また、異なるシミュレータでは異なる結果を生成します。

```
 1  module test;
 2  reg      clk = 0,
 3           _reset = 0;
 4
 5  dut DUT(clk,_reset);
 6  // ...
 7  endmodule
 8
 9  module dut(input wire clk,_reset);
10           always @(posedge clk or negedge _reset) begin
11                    // ...
12           end
13  // ...
14  endmodule
```

Verilog HDLの問題点（イベント制御機能）

- イベント待ち（@ev）はエッジ・センシティブですが、evには値が無いので状態は残りません。この為、@ev待ちのプロセスがイベント解除をする時期を見逃してしまう可能性があります。

Verilog で次の様な記述をすると何れのinitialプロシージャが先に実行しても、@ev1、又は、@ev2のどちらかが解除されません。

```
 1  module test;
 2  event   ev1, ev2;
 3          initial begin
 4                  ->ev2;
 5                  @ev1;
 6                  $display("@%0t: ev1 released.",$time);
 7          end
 8          initial begin
 9                  ->ev1;
10                  @ev2;
11                  $display("@%0t: ev2 released.",$time);
12          end
13  endmodule
```

新たに追加されたイベント制御機能 (triggered()メソッド)

- SystemVerilogでは、@ev1、及び、@ev2の代わりに、triggered()を使用します。この関数は状態を保存している為、イベントが発生した事を確実に捉える事が出来ます。
- メソッドtriggered()はエッジ・センシティブな状態をレベル・センシティブな状態に変換をします。

次の記述で何れのイベント待ちも正しく解除されます。

```
1  module test;
2  event   ev1, ev2;
3          initial begin
4                  ->ev2;
5                  wait ( ev1.triggered() );
6                  $display("@%0t: ev1 released.",$time);
7          end
8          initial begin
9                  ->ev1;
10                 wait ( ev2.triggered() );
11                 $display("@%0t: ev2 released.",$time);
12         end
13 endmodule
```

豊富なデータタイプ

- SystemVerilogには2-stateタイプ（bit、byte、shortint、int、longint）が追加されました。これらのタイプは、0で初期化されます。
- SystemVerilogは豊富なデータタイプを持ちます。2-stateタイプの他に、例えば、package、interface、class、enum、struct、union、string、mailbox、semaphore、program等があります。
- Automatic属性が追加されました。タスク、ファンクション、及び、変数にautomatic属性を付加する事が出来ます。
- typedef文により、ユーザ独自のデータタイプを定義する事が出来ます。
- Verilog HDLと異なり、変数を2次元以上のアレイとして定義する事が出来ます。
- アレイのサイズを実行時に決定する事も出来ます。これらの例としては、dynamic arrays、queues、associative arraysがあります。
- アレイの要素を操作する為のメソッドが多く存在します。
- SystemVerilogではアレイの初期化も容易です（アレイ・リテラル、associativeアレイ・リテラル、キュー・リテラル）。

2-Stateタイプ（使用例）

```
 1  class ubus_transfer;
 2  rand bit [15:0]             addr;
 3  rand ubus_read_write_enum read_write;
 4  rand int unsigned           size;
 5  rand bit [7:0]              data[];
 6  rand bit [3:0]              wait_state[];
 7  rand int unsigned           error_pos;
 8  rand int unsigned           transmit_delay = 0;
 9  string                      master = "";
10  string                      slave = "";
11
12  constraint c_read_write {
13          read_write inside { READ, WRITE };
14  }
15  constraint c_size {
16          size inside {1,2,4,8};
17  }
18  constraint c_data_wait_size {
19          data.size() == size;
20          wait_state.size() == size;
21  }
22  constraint c_transmit_delay {
23          transmit_delay <= 10;
24  }
25  endclass : ubus_transfer
```

C/C++と異なり、unsignedをintの後に指定をする。

検証で使用するトランザクションでは、2-stateタイプを多く使用します。

Dynamic Arrays and Associative Arrays（Associative Arrayの高度な使用例）

```
 1  package Pkg;
 2  string  PA[process];     // associative array on process whose value is process name.
 3  string  Pstatus[] = '{ "FINISHED", "RUNNING", "WAITING", "SUSPENDED", "KILLED" };
 4  class X;
 5
 6  static function void create(int n);
 7  /*
 8          Create n processes.
 9  */
10  PA.delete;
11  for( int i = 0; i < n; i++ )
12          fork    automatic int k = i;
13                  automatic process p = process::self();
14                  begin
15                          PA[p] = $sformatf("process-%0d",k);
16                          $display("@%2t: %s suspending...",$time,PA[p]);
17                          p.suspend;
18                          $display("@%2t: %s resumed",$time,PA[p]);
19                  end
20          join_none;
21  endfunction
22  // ...
23  endclass
24  // ...
25  endpackage
```

ダイナミック・アレイを宣言時に初期化する事が出来る。

Associative array PA[process]を構築している。

Static and Automatic Variables

- Verilog HDLでは、全ての変数はstaticです。即ち、同じ名称を持つ変数は唯一つしか存在しません。この為、recursiveコール等の処理を記述する事は困難を伴いました。
- SystemVerilogでは、全ての変数に対して、static、又は、automaticの属性を定義する事が出来ます。Automatic変数を使用すると、recursiveコールの記述が容易になります。

```
 1  module test;
 2
 3          initial begin
 4                  for( int i = 0; i <= 7; i++ ) begin
 5                          $display("%0d! = %0d",i,factorial(i));
 6                  end
 7          end
 8
 9  function automatic int factorial(int n);
10          if( n < 2 )
11                  return 1;
12          else
13                  return n*factorial(n-1);
14  endfunction
15  endmodule
```

Automaticを使用するとrecursive callを簡潔に記述する事が出来る。

fork/join
（実践で必要とする重要な機能）

- 並列処理を記述する際に使用する機能が追加されました。Verilog HDLにはfork/joinしかありませんでした。
- SystemVerilogには、fork-join、fork-join_any、fork-join_noneの三種類があります。forkブロック内の文は同時に実行します。ブロック内の文は子プロセス（スレッド）として実行します。対のキーワードにより機能が以下の様に異なります。

対のキーワード	意味
join	全ての子プロセスが終了するまで、親プロセスは待ち状態に入ります。
join_any	何れか一つの子プロセスが終了するまで、親プロセスは待ち状態に入ります。
join_none	全ての子プロセスを生成しますが、親プロセスは実行を継続します。生成された子プロセスは、親プロセスがブロックするか終了するまで実行を開始しません。

fork/join
（例）

```
 1  module test;
 2
 3          initial begin
 4                  fork
 5                          #10 print(1);
 6                          #20 print(2);
 7                          #30 print(3);
 8                  join
 9
10                  $display("@%2t main completed.",$time);
11          end
12
13  function void print(int value);
14          $display("@%2t value=%0d", $time, value);
15  endfunction
16
17  endmodule
```

```
@10 value=1
@20 value=2
@30 value=3
@30 main completed.
```

fork/join_any（例）

```
 1  module test;
 2
 3          initial begin
 4                  fork
 5                          #10 print(1);
 6                          #20 print(2);
 7                          #30 print(3);
 8                  join_any
 9
10                  $display("@%2t main completed.",$time);
11          end
12
13  function void print(int value);
14          $display("@%2t value=%0d", $time, value);
15  endfunction
16
17  endmodule
```

```
@10 value=1
@10 main completed.
@20 value=2
@30 value=3
```

fork/join_none（例）

```
 1  module test;
 2
 3          initial begin
 4                  fork
 5                          print(1);
 6                  #20 print(2);
 7                  #30 print(3);
 8                  join_none
 9
10                  $display("@%2t main completed.",$time);
11          end
12
13  function void print(int value);
14          $display("@%2t value=%0d", $time, value);
15  endfunction
16
17  endmodule
```

```
@ 0 main completed.
@ 0 value=1
@20 value=2
@30 value=3
```

Forkブロック関する注意
（細心の注意が必要なケース）

- ブロック内で生成された子プロセスは親プロセスがブロックするまで実行を開始しません。この為、変数の参照には特別な注意が必要です（[1],[3]）。
- 次の例には、forループの内部にfork-join_noneがあり、ループ制御変数iの値をプリントしています。iの値が123とプリントされる事を期待しているコーディングですが、予想に反して結果は444になります。

```
for( int i = 1; i <= 3; i++ )
        fork
                $write("%0d",i);
        join_none
#0 $display;
```

- fork-join_noneにより$writeのプロセスが生成されますが、親プロセスがブロックするまで、子プロセスは開始しません。forループが終了した時に親プロセスはブロックしますが、この時にはi==4となっています。従って、三つの$writeのプロセスは4をプリントします。

Forkブロック関する注意（正しい記述法）

- forkブロック内の子プロセスは親プロセスと異なるタイミングで実行する為、現在の実行環境を保存しておく必要があります。
- 実行環境を保存する為に、新たな変数をautomaticで確保して、以下の様に定義します。こうすると、123とプリントされます。

```
for( int i = 1; i <= 3; i++ )
        fork automatic int k = i;
                $write("%0d",k);
        join_none
#0 $display;
```

- fork文が実行する時、iの値がkに保存されます。kはautomaticである為、ループ回数だけkが確保されます。即ち、三つの子プロセスは独自のkを保有する事になります。それらは、順に1、2、3です。

プログラミング機能

- Verilog HDLに対して、SystemVerilogは多くの機能を追加しました。例えば、
  - 値を戻さないファンクションを記述する事が出来ます。
  - タスク、及び、ファンクションの引数に標準値を設定する事が出来ます。
  - Always文には幾つかの種類が追加されました。
  - 便利なforeach文が追加されました。
  - プロシージャのループ内で、continue、及び、break文を使用出来ます。
  - Return文により、処理を途中で終了する事が出来ます。
  - Do while文があります。
  - Unique-if、及び、unique-case文が追加されました。
  - SystemVerilogは、便利な演算子（++、--、+=、等）を備えています。
  - 等等。

プロセス間通信機能 (Inter-Process Communication:IPC)

- 並列して実行しているプロセス間で同期を取る手段として、event、mailbox、及び、semaphoreがあります。
- セマフォは、複数のプロセスが共有する資源にアクセスする為の排他制御機能です。セマフォはSystemVerilogのクラスとして実現されています。
- メール・ボックスはproducerとconsumerをもつFIFOリストです。FIFOリストは限られたサイズ、又は、無限のサイズを持つ様に定義する事が出来ます。
- eventデータ型を持つ変数は名称付きイベントとなります。名称を使用してイベントの操作を行います。イベントが発生するまで、実行をブロックする事が出来ます。

検証機能

- 近年のテスト法は、CRT（Constrained Random Tests）をベースしています。CRTは、制約を満たすテスト・データをランダムに生成する技術です。CRTはSystemVerilogの機能として実現されています。
- CRTによりテスト・データを生成すると、テスト結果を自動的に集計しなければなりません。SystemVerilogのファンクショナル・カバレッジは、カバレッジ集計機能です。
- アサーションは、仕様とデザイン（仕様を実装した内容）が一致する事をチェックする機能です。一致しない場合、明瞭なレポートを作成する事が出来ます。アサーションはSystemVerilogの重要な機能として実現されています。

ファンクショナル・カバレッジ

- 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。
- 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。
- SystemVerilogファンクショナル・カバレッジでは、信号値だけでなく信号の値の遷移（transitions）の計測も行う事ができます。
- SystemVerilogでは、ファンクショナル・カバレッジの仕様をソース・コード中に記述する事が出来ます。しかも、記述されたカバレッジ仕様はシミュレータにより実行されます。

カバレッジ・モデル（定義例）

```
 1  class Transaction;
 2          rand bit [3:0]  kind;
 3          rand bit [2:0]  port;
 4  endclass
 5
 6  program automatic test;
 7  Transaction     tr;
 8
 9  covergroup CovPort;
10          option.per_instance = 1;
11          kind:   coverpoint      tr.kind
12                  {
13                          bins    zero = {0};
14                          bins    lo = { [1:3] };
15                          bins    hi[] = { [8:$] };
16                          bins    misc = default;
17                  }
18          port:   coverpoint      tr.port
19                  {
20                          bins port[] = { [0:$] };
21                  }
22          cross   kind, port;
23  endgroup
24  // ...
25  endprogram
```

SystemVerilogアサーション（SystemVerilog Assertions：SVA）

- アサーションはシステムの動作（即ち、仕様）を記述する為の手段です。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、functional coverage、又は、入力となるスティムラスの検証にも使用されます。
- アサーションではシーケンス（sequences）、及び、プロパーティ（properties）を用いて仕様を記述します。但し、それらの記述自身では起動しない為、assert、cover、assume等のアサーション文を使用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。

検証 ::=（仕様 == デザイン） ？ pass_statement ：fail_statement;

- 検証者は仕様と合否処理（pass_statementとfail_statement）を記述します。その他の全ての検証タスクはシミュレータが担当します。デザインをシミュレーションし、（仕様 != デザイン）となる状況が発生するとfail_statementが実行します。
- SystemVerilogアサーションの特徴は、仕様とデザインの不一致があれば、デザインの何処に問題があるかを容易に特定する事が出来る事です。

アサーションの種類

- アサーションには、即時実行型（immediate）と並列型（concurrent）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。従って、時間を消費する概念がありません（non-temporal）。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.");
// ...
```

- この例では、クラスpacket_tのメンバーに乱数が正しく割り当てられているかを確認しています。メンバーには制約が課されている為、全ての制約を満たす様に乱数を発生する事が成功するとは限りません。その為、assert文を使用して確認をしています。乱数発生が不成功の場合、$fatal()タスクでエラーを報告します。

即時実行型アサーション（使用例）

- 即時実行型アサーションの例を示す為に interface を使用します。このinterfaceの役目は、入力信号の正当性を確認する事です。その目的でimmediate assert文を使用します。interfaceを以下の様に定義します。

```
1  interface simple_if(input bit clk,input logic set,reset);
2
3  modport TEST(input clk,output set,reset);
4
5  always @(posedge clk)
6          assert( !$isunknown(set) && !$isunknown(reset) )
7          else $error("@%0t: FAIL set=%b reset=%b",$time,set,reset);
8
9  endinterface
```

- @(posedge clk)のイベントが起こる度に、set及びresetの正当性チェックが行われます。
- 即時実行型アサーションの機能は比較的自明なので、以降では並列型アサーションを主として解説します。

並列型アサーション (concurrent assertions)

- 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。記述した検証条件が成立、又は、不成立するまで検証が続行します。
- 実行はクロックと同期して開始、又は、再開します。従って、一つのアサーション記述に対して複数のインスタンスが同時に進行している可能性がある為、並列型アサーションはマルチ・スレッドとなります.
- 並列型アサーションはクロック・サイクルをベースにする為、サイクル・ディレー（## オペレータ）を使用して動作を記述します。
- 並列型アサーションでは、クロックの他に大切な概念があります。それは、アサーションで使用する信号の値です。アサーションでは、どの様なタイミングで信号値を取得するかが重要になります。この信号値はサンプル値（sampled values）と呼ばれ、SystemVerilogはサンプル値を厳密に定義しています。
- 並列型アサーションの評価はobserved regionで行われます。

並列型アサーション（使用例）

- 次の例は並列型アサーションの一例です。この例では、default clocking を使用しています。

```
module test(input clk,req,ack);
        default clocking cb @(posedge clk); endclocking
        sequence check_req_ack;
                req ##[1:3] ack;
        endsequence

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

- サイクル・ディレーは @(cb) 、即ち、@(posedge clk) により決定されます。@(cb)のイベントが起こるタイミングでサンプル値が取得されて、次の条件が満たされる事を確認します。

```
req ##[1:3] ack;
```

- req==1’b1であれば、1~3クロック・サイクルの間にack==1’b1が成立しなければなりません。成立しない場合、エラー・メッセージがプリントされます。

データタイプ

SystemVerilogは多くのデータタイプを持ちますが、この章ではstring、enum、struct、union等のデータタイプを中心に解説します。

String Types
（実践で必要とするデータタイプ）

- String型の変数には以下の様な演算を適用する事が出来ます。特に、文字列の結合オペレータ{}は非常に便利なオペレータです。Stringはクラスとして実現されています。従って、methodsが定義されています。

演算子	意味
str1 == str2	二つの文字列が等しければ1になります。等しくない時には0と評価されます。
str1 != str2	str1 == str2の否定形です。
str1 < str2 str1 <= str2 str1 > str2 str1 >= str2	二つの文字列を辞書引き順序で比較し、正しければ1を戻し、偽の場合には0を戻します。
{str1,str2,…,strn}	文字列を結合します。
{multiplier{str}}	文字列を指定した回数繰り返します。
str[index]	indexで指定した文字（byte型）を戻します。indexは0からN-1の範囲でなければなりません。ここで、Nは文字列の長さを示します。
str.method(…)	string型に定義されているmethodを実行します。

String Type's Methods （その1）

method	意味
function int len();	str.len()は文字列strの長さを戻します。空文字列の長さは0です。
function void putc(int i, byte c);	str.putc(i,c)はstr[i]=cと同じです。
function byte getc(int i);	x=str.getc(i)はx=str[i]と同じです。
function string toupper();	str.toupper()は大文字に変換した文字列を戻します。strの内容は変わりません。
function string tolower();	str.tolower()は子文字に変換した文字列を戻します。strの内容は変わりません。
function int compare(string s);	str.compare(s)はC/C++のstrcmp(str,s)と同じです。
function int icompare(string s);	str.icompare(s)はC/C++のstricmp(str,c)と同じです。
function string substr(int i, int j);	str.substr(i,j)はインデックスiからjまでの部分文字列を戻します。
function integer atoi(); **function integer atohex();** **function integer atooct();** **function integer atobin();**	str.atoi()は整数を表現している文字列が表す10進整数を戻します。例えば、strが”123”であれば、str.atoi()は123を戻します。 atohex()は16進数、atooct()は8進数、atobin()は2進数です。

String Type's Methods （その2）

method	意味
function real atoreal();	str.atoreal()はstrが表現する実数を戻します。例えば、strが ”3.14” であれば、str.atoreal()は3.14を戻します。
function void itoa(integer i); **function void hextoa(integer i);** **function void octtoa(integer i);** **function void bintoa(integer i);** **function void realtoa(real r);**	integer iを文字列に変換してstrに設定します。 他のfunctionも同様です。

- Stringはクラスですが、newメソッドが定義されていないので、newオペレータでオブジェクトを作る事は出来ません。

String 使用例
（演算子==、toupper()、文字列の結合）

- 文字列の結合機能は実践で頻繁に使用される便利な機能です。

```
1  module test;
2  string  lang = "SystemVerilog";
3  string  ieee = "ieee";
4
5  initial begin
6          if( lang == "SystemVerilog" )
7                  $display("%s",{ieee.toupper()," Std 1800-2017"});
8  end
9  endmodule
```

```
IEEE Std 1800-2017
```

Enumデータタイプ

- Enumデータ・タイプは関連する値を持つコンスタントを定義する命令です。値を明示的に設定する事が出来ますが、自動的に設定させる事も出来ます。
- C++のenum機能と類似していますが、異なる点もあります。例えば、enumデータ・タイプにintegral type（logic、bit、byte、shortint, int等々）を指定する事が出来ます。
- また、enumデータ・タイプには標準的なmethodsが定義されています。タイプが省略されると、int型が仮定されます。

```
1  typedef enum {GREEN, YELLOW, RED} controller_e;
2
3  module test;
4  controller_e    light1 = GREEN,
5                  light2 = RED;
6  // ...
7  endmodule
```

Enum変数を操作する関数（Enum Methods）

- enumには便利なmethodが定義されています。

method	意味
function enum first();	最初のenumラベルを戻します。
function enum last();	最後のenumラベルを戻します。
function enum next(int unsigned N = 1);	現在よりN個先のラベルを戻します。最後のラベルの次は最初に戻ります。
function enum prev(int unsigned N = 1);	現在よりN個前のラベルを戻します。最初のラベルに行くと最後のラベルに戻ります。
function int num();	Enumに定義されているラベル数を戻します。
function string name();	現在のenumラベル名を戻します。

Enum変数を操作する関数（使用例）

```
1  typedef enum { GREEN=1, YELLOW=3, RED=10 } color_e;
2
3  module test;
4  color_e c;
5
6          initial begin
7                  c = c.first;
8                  forever begin
9                          $display("%s=%0d",c.name,c);
10                         if( c == c.last )
11                                 break;
12                         c = c.next;
13                 end
14         end
15
16 endmodule
```

```
GREEN=1
YELLOW=3
RED=10
```

Structデータタイプ

- Structureは幾つかのメンバーを一つのグループとして纏めたデータ・タイプです。各メンバーは異なるデータ・タイプを持つ事が出来ます。メンバーを個別に参照、又は、操作する事が出来ます。グループ全体として参照する事も出来ます。
- Structureにはpacked structureとunpacked structureがあります。unpacked structureがdefaultで、修飾子unpackedを省きます。
- packed structureではメンバー全体が連続したメモリーに配置されます。従って、structure全体を一つの数値として操作する事が出来ます。
- unpacked structureの場合には、メンバーが連続したメモリーに配置される保証はありません。

Structデータタイプ（unpacked structの使用例）

```
 1  struct { bit [7:0] r, g, b; }                    pixel;
 2  typedef struct { bit [7:0] r, g, b; }            pixel_s;
 3  typedef struct { string name; int price; }       food_s;
 4
 5  module test;
 6  pixel_s px;
 7  food_s  fd;
 8
 9          initial begin
10                  pixel.r = 0;
11                  pixel.g = 255;
12                  pixel.b = 255;
13
14                  px.r = 255;
15                  px.g = 100;
16                  px.b = 64;
17
18                  $display("pixel=(%0d,%0d,%0d) px=(%0d,%0d,%0d)",
19                          pixel.r,pixel.g,pixel.b,px.r,px.g,px.b);
20
21                  fd.name = "bread";
22                  fd.price = 250;
23                  $display("food=(%s,%0d)",fd.name,fd.price);
24          end
25  endmodule
```

Structデータタイプ（packed structの使用例）

```
 1  struct packed signed {
 2          int a;
 3          shortint b;
 4          byte c;
 5          bit [7:0] d;
 6  } pack1;
 7
 8  program automatic test;
 9
10          initial begin
11                  $display("$bits(pack1)=%0d",$bits(pack1));
12                  pack1.b = -1;
13                  pack1[15:8] = 8'hab;            // access to c
14                  pack1[62] = 1;
15                  $display("pack1=%h pack1[15:8]=%h",pack1,pack1[15:8]);
16          end
17
18  endprogram
```

Structの最初のメンバーはMSBに位置し、最後のメンバーはLSBに配置されます。

63	16	8	0
a(32ビット)	b(16ビット)	c(8ビット)	d(8ビット)

Unionデータタイプ

- Unionは一つのオブジェクトを異なる名称で参照する事が出来るデータ・タイプです。structuresと同様に、packed unionの概念があります。structuresとの違いは、メンバーが同じ領域を共有する事にあります。
- 領域を共有する事が目的である為、各メンバーのデータ・タイプは異なります。一見、メモリー使用量の削減効果を持つ様に思えますが、実質的には実行上のオーバーヘッドが大きいという短所を持っています。特別な目的を持たないのであれば、unionは使用を避けるべきデータ・タイプの一つであると思えます。
- 因みに、Javaはunionデータタイプの機能を備えていません。

Typedef

- ユーザ固有のデータタイプを定義する事が出来ます。

```
1  parameter OPERAND_WIDTH = 4;
2  parameter OPERATOR_WIDTH = 3;
3
4  typedef enum logic [OPERATOR_WIDTH-1:0]
5          { ADD=3'b001, SUB=3'b010, XOR=3'b100 } opcode_e;
6
7  typedef bit [31:0]      uint;
8
```

- この機能を利用すると、まだ存在しないデータ・タイプを暫定的に定義する事が出来ます。相互に参照しあう二つのデータ・タイプを定義する際には、この機能が必要になります。この機能をforward typedef宣言と呼びます。C++のforward declarationに相当します。

アレイの操作

この章ではアレイの種類とアレイを操作する為のメソッドを解説します。また、アレイの初期化機能についても触れます。これらの機能についての知識は簡潔なコードを書く為に必要です。

Queues、及び、associativeアレイは非常に大切な概念である為、詳しく解説します。UVMのソース・コードを解析する際には、必要な知識となります。

アレイ

- Verilogと異なり、多次元のアレイを定義する事が出来ます。
- アレイには、キュー、associative arrayもあります。
- sarray[] はダイナミック・アレイの例です。
- price[string] アレイはassociative arrayの例です。
- foods[$]、及び、data[$] はキューの例です。

```
 1  module test;
 2  int      a[1:2][1:3] = '{ '{0,1,2}, '{3{4}} };
 3  enum { January=1, February, March, April, May, June, July,
 4         August, September, October, November, December } month;
 5  int      days[1:12] = '{ 2:28, 4:30, 6:30, 9:30, 11:30, default:31 };
 6  string   your_foods[5] = '{ "bread", "milk", "rice", "fish", "meat" };
 7  shortint         sarray[];
 8  int      price[string] = '{ default : 100 };     // associative array
 9  string   foods[$] = { "bread", "milk", "meat", "rice", "noodle" };
10  int      data[$] = { 1, 2, 3 };
11
12  initial begin
13          sarray = new[5];
14          // ...
```

アレイ・リテラル

キュー・リテラル

packed アレイとunpackedアレイ（packed and unpacked arrays)

- SystemVerilogでは、名称の前に位置する次元定義をpacked array、名称の後に位置する次元定義をunpacked arrayと呼びます。例えば、

```
bit [1:0][7:0] TwoBytesArray [20];
```

- の定義に於いて、[1:0][7:0]はpacked array、[20]はunpacked arrayです。この場合のpacked arrayは二次元ですが、一次元のpacked arrayはvectorとも呼ばれます。

Packed アレイ（packed arrays)

- Packed arraysはvectorを階層的に操作する為の宣言手段です。Packed arraysのビットは連続領域にとられる事が保障されています。例えば、

```
bit [2:1][7:0]  bytes;
```

- に於いて、bytesは2バイトの領域を確保しています。bytes[2]とbytes[1]は物理的に連続した領域に存在します。

bytes															
bytes[2]								bytes[1]							
[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]

- 全体をbytesとして参照する事が出来ます。最上位のビットをbytes[2][7]、最下位のビットをbytes[1][0]として参照する事が出来ます。
- bytes[2]はビット位置としてbytes[1]よりも高位の位置にあります。

Unpacked アレイ（unpacked arrays)

- Unpacked arraysはpacked arraysと異なり不連続なアレイです。即ち、アレイの要素が連続した場所に存在する保障はありません。連続していない為に、unpackedと呼ばれます。例えば、

```
bit [7:0]       name[3:0];
```

- に於いて、nameは4バイトの大きさを持っています。name[0]、name[1]、name[2]、name[3]の様に参照する事が出来ますが、それらのバイトは連続していません。

- Unpacked arraysはサイズを指定するだけで宣言を済ませる事が出来ます。例えば、次の二つの宣言は同じです。

```
shortint        matrix[0:7][0:127];
shortint        matrix[8][128];
```

- 即ち、unpacked arrayの宣言に於いて、[N]は[0:N-1]と同じ効果を持ちます。packed arraysに対してこの記法を使用する事は出来ません。

アレイの走査
（foreach文の使用例）

- SystemVerilogにforeach文が追加されました。For文でループ処理を記述するよりも効率良く処理をする事が出来ます。
- 下記の例に於いて、ループ制御変数 i は自動的に確保されます。この変数は、ローカルに確保され、しかも、automaticです。

```
 1  module test;
 2  enum { January=1, February, March, April, May, June, July,
 3         August, September, October, November, December } month;
 4  int    days[1:12] = '{ 2:28, 4:30, 6:30, 9:30, 11:30, default:31 };
 5
 6         initial begin
 7                 foreach(days[i]) begin
 8                         $cast(month,i);
 9                         $display("%s=%0d",month.name,days[i]);
10                 end
11         end
12
13 endmodule
```

```
January=31
February=28
March=31
...
```

ダイナミック・アレイ（使用例）

- ダイナミック・アレイの割り当てには、newオペレータを使用します。

```
 1  module test;
 2  shortint        sarray[], v10[], copy_s[],
 3                  fixed_s[6] = { 1, 2, 3, 4, 5, 6 };
 4
 5          initial begin
 6                  sarray = new[5];                    // (0,0,0,0,0)
 7                  print("sarray",sarray);
 8
 9                  v10 = new[5](10);                   // (10,10,10,10,10)
10                  print("v10",v10);
11
12                  copy_s = new[fixed_s.size()](fixed_s); // (1,2,3,4,5,6)
13                  print("copy_s",copy_s);
14          end
15
16  function void print(string msg,shortint a[]);
17          $write("%s=",msg);
18          foreach(a[i])
19                  $write(" %0d",a[i]);
20          $display;
21  endfunction
22  endmodule
```

キュー

- Queuesは伸縮自在なアレイです。データ構造のstacks、及び、queuesとして使用する事が出来ます。
- ダイナミック・アレイと異なり、宣言した時点でキューが有効になります。即ち、要素数が0のキューが作成されます。
- Queuesでは、アレイと同じ様に整数型インデックスにより要素にアクセスします。最初の要素はインデックスが0です。最後の要素のアクセスにはインデックス $ を使用します。
- Queuesを空にする為に、{ }を使用する事が出来ます。
- Queuesにはアレイと同様の演算を適用する事が出来ます。アレイと異なる点は、queuesは伸縮自在である事です。キュー qに対してq[a:b]もキューです。

キューを操作する関数（queue methods）

メソッド	意味
function int size();	Queueの要素数を戻します。
function void insert(input integer index, input element_t item);	指定した位置（index）に要素を挿入します。位置が正しくない場合、何も挿入されません。
function void delete([input integer index]);	indexが指定された場合、そのindex位置にある要素を削除します。 indexが省略された場合、queueの全ての要素を削除します。
function element_t pop_front();	Queueの先頭の要素を削除して戻します。
function element_t pop_back();	Queueの最後の要素を削除して戻します。
function void push_front(input element_t item);	Queueの先頭に要素を挿入します。
function void push_back(input element_t item);	Queueの最後に要素を追加します。

キューを操作する関数（使用例）

- 操作関数を以下の様に簡単に使う事ができます。

```
 1  module test;
 2  int     q[$] = { 1, 4, 6 };
 3
 4          initial begin
 5                  print("q",q);           // {1,4,6}
 6                  q.push_front(0);
 7                  q.push_back(7);
 8                  print("a",q);           // {0,1,4,6,7}
 9                  q.insert(2,2);
10                  print("q",q);           // {0,1,2,4,6,7}
11                  q.delete(0);
12                  print("q",q);           // {1,2,4,6,7}
13                  q.delete;
14                  print("q",q);           // {}
15          end
16  function void print(string msg,int a[$]);
17          $write("%s:",msg);
18          foreach(a[i])
19                  $write(" %0d",a[i]);
20          $display;
21  endfunction
22  endmodule
```

Associativeアレイのインデックス・タイプ

[index_type]	意味
[*]	Wildcard indexタイプと呼ばれ、任意の整数式をインデックスとして使用する事が出来ます。但し、four stateの値をインデックスとして使用する事は出来ません。
[string]	インデックスはstringタイプです。空文字列もキーとして許されます。辞書引き順序が使用されます。
[class]	クラス・オブジェクトをインデックスとします。オブジェクトはC++でいうポインターに相当するので、インデックスとしてナル（null）も許されます。キーの順序は定まっていますが、実装するシミュレータにより異なります。一般的には、オブジェクトのアドレスがキーになっていると考えて良いと思います。
[integer] [int] [shortint] [byte] [longint]等	整数系のインデックスを使用します。但し、four stateの値をインデックスとして使用する事は出来ません。

Associativeアレイの要素の登録 (使用例)

- priceはstring型をキーに持つassociative arrayです。定義に於いて標準値を100に設定しています。もしキーで示すデータが存在しない場合には整数値100をデータとしてpriceに登録します。
- 最初はprice.size==0である為、price[key]は存在しません。定義により、price[key]の要素が作られ、price[key]==100となります。price[key]++の実行後に、price[key]は101になります。詰まり、price[“bread”]==101です。
- その後、price[key] += 20によりprice[“bread”]==121となります。

```
1  module test;
2  int     price[string] = '{ default : 100 };
3  string  key;
4
5          initial begin
6                  key = "bread";
7                  price[key]++;                          // 101
8                  $display("%s=%0d",key,price[key]);
9
10                 price[key] += 20;                      // 121
11                 $display("%s=%0d",key,price[key]);
12         end
13 endmodule
```

Associativeアレイ・リテラル（ Associativeアレイの初期化機能）

- Associative arraysの初期化にはアレイ・リテラル '{...} を使用します。
- 初期化には、キーとデータ値の対をコロン(:)で区切って指定します。ここでは指定をしていませんが、default値も与える事が出来ます。

```
 1  module test;
 2  integer work_hours[string] = '{
 3          "Sunday":0, "Monday":8, "Tuesday":10,
 4          "Wednesday":10, "Thursday":8,
 5          "Friday":8, "Saturday":0 };
 6  string  key;
 7
 8          initial begin
 9                  foreach(work_hours[key])
10                          $display("%s=%0d",key,work_hours[key]);
11          end
12  endmodule
```

クラス
(Classes)

クラスは関連する要素を同じグループに纏め、それらの要素を操作する機能を提供する便利なデータタイプです。検証作業には欠かせないツールです。

UVMの様な検証パッケージを使用する為には、クラスに関する知識を完全にマスターする事が必要です。

Class

- クラスはデータ・タイプの一つで、データとサブルーティン（functions、及びtasks）から構成されます。クラスでは、データをプロパーティ（properties）、サブルーティンをメソッド（methods）と呼びます。
- クラスにはnewと呼ばれる特別なメソッドがあり、コンストラクタと呼びます。コンストラクタはクラスのインスタンス（つまり、オブジェクト）を創成します。
- SystemVerilogのクラスの概念はC++のそれよりJavaのクラスに似ています。
- 例えば、サブクラスを定義する為には、Javaと同様にextendsを使用します。ダイナミックに割り当てたメモリーの解放は、Javaと同様に自動的に行なわれます。
- クラスのメソッドには重要な属性virtualがあります。これは、クラス・インヘリタンス（継承）に従って、対応するメソッドが呼び出される機能です。

Class
（定義例）

```
 1  package Pkg;
 2  parameter       ADDR_WIDTH = 32;
 3  parameter       DATA_WIDTH = 32;
 4  typedef logic[ADDR_WIDTH-1:0]   addr_t;
 5  typedef logic[DATA_WIDTH-1:0]   data_t;
 6  // simple_transaction
 7  class simple_transaction;
 8  static int      counter;
 9  local string    name;
10  addr_t          addr;
11  data_t          data;
12  // constructor
13  function new(string name,addr_t a=100,data_t d=0);
14          this.name = name;
15          addr = a;
16          data = d;
17          counter++;
18  endfunction
19  // virtual method
20  virtual function void print;
21          $display("name=%s addr=%h data=%h",name,addr,data);
22  endfunction
23  // ...
24  endclass
25  endpackage
```

クラス・オブジェクトとメンバーへのアクセス

- クラスはデータ・タイプです。オブジェクトはクラスのインスタンスです。クラス・タイプを持つ変数を定義して、次の様に変数にオブジェクトを割り当てます。

```
simple_transaction      tr = new("TR1");
```

- SystemVerilogではこの変数をオブジェクト・ハンドルと呼びます。オブジェクトを割り当てていないハンドルにはナル（null）が割り当てられます。ナルのハンドルでプロパーティやメソッドにアクセスすると、一般的には、異常終了すると考えた方が安全です。
- SystemVerilogでは、プロパーティ及びメソッドへのアクセスにはハンドルとドット（.）を使用します。C/C++の様な記法（->）は出来ません。

```
tr.print();
```

- Staticなメンバーへのアクセスは、スコープ・オペレータを使用する事が出来ます。

```
simple_transaction::counter
```

クラス・インヘリタンス（クラスで最も重要な概念）

- 既に定義済みのクラスを使用して、新しいクラスを定義する事が出来ます。新しいクラスはサブクラスと呼ばれます。基のクラスをベース・クラスと言います。
- Javaと同様に、キーワード extends を使用してサブクラスを定義します。
- 既存のクラスをベースにして新しいクラスを定義する場合には、コンストラクタの最初の命令はsuper.new()でなければなりません。
- サブクラスでは、ベース・クラスの local メンバー以外のメンバーを自由に使用する事が出来ます。

クラス・インヘリタンス（使用例）

- コンストラクタの最初の命令は super.new(name,a,d) になっています。
- Virtual メソッド print の内容を書き換えています。
- クラスには、自分自身を参照する為の特別なハンドル this があります。

```
25  class delayed_simple_transaction extends simple_transaction;
26  int     delay;                          ← ベース・クラスにメンバーを追加。
27
28  function new(string name,addr_t a=100,data_t d=0,int delay=0);
29          super.new(name,a,d);
30          this.delay = delay;             ← this ハンドル
31  endfunction
32
33  function void print;
34          super.print();
35          $display("delay=%0d",delay);
36  endfunction
37
38  endclass
```

StaticとAutomaticプロパーティ（必須の知識）

- クラスに定義したメンバーは、原則として、automaticです。従って、クラス・インスタンス毎に存在します。しかも、メソッドが呼ばれると引数、及び、メソッド内の変数はスタック上に確保されます。言い換えると、以前の状態は保存されていません。
- 同じクラス・タイプのインスタンス全てに共通なメンバーを定義する為には、staticという修飾子を使用します。プロパーティ、及び、メソッドに対してこの修飾子を適用する事が出来ます。
- staticメンバーは唯一つしか存在しないという性質から、クラス・スコープを使用してstaticメンバーへのアクセスをする事が出来ます。クラスのインスタンスを作る必要はありません。

```
simple_transaction::counter
```

Virtual Methods
（実践で必要となる重要な技術）

- C++でいうvirtual functionsと同じ概念です。ベース・クラスにvirtualメソッドvmを定義すると、継承されたサブクラスのオブジェクトに一致するメソッドvmを呼び出します。
- 言い換えると、クラス・ハンドルがベース・クラスのタイプでも、ハンドルが継承されたクラスのインスタンスを示していれば、継承されたクラスのvmを呼び出します。
- 尚、Javaにも対応する概念がありますが、Javaではキーワードvirtualを使用しません。この点は、JavaとSystemVerilogの言語上の差異です。

Virtual Methodsの使用例（この技術の習得はMUSTです）

- base_t::print()はvirtualです。サブクラスtr_tではbase_t::print()を再定義しています。メソッドprint_class()が両方のクラスに定義されていますが、これらは無関係です。

```
 1  class base_t;
 2  virtual function void print;
 3          // ...
 4  endfunction
 5  function void print_class;
 6          // ...
 7  endfunction
 8  endclass
 9  class tr_t extends base_t;
10  // ...
11  function new(bit [15:0] a=0,d=0);
12          // ...
13  endfunction
14  function void print;
15          // ...
16  endfunction
17  function void print_class;
18          // ...
19  endfunction
20  endclass
```

```
22  module test;
23  base_t  base;
24  tr_t    tr;
25
26  initial begin
27          base = new;
28          tr = new(10,50);
29          base.print;         // base_t::print
30          base.print_class;   // base_t::print_class
31          tr.print;           // tr_t::print
32          tr.print_class;     // tr_t::print_class
33          base = tr;
34          base.print;         // tr_t::print
35          base.print_class;   // base_t::print_class
36  end
37  endmodule
```

アブストラクト・クラスと
ピュア・バーチャル・メソッド

- クラスを継承する事により、クラスの定義をより詳細に定義して行きます。その際、ベース・クラスの決めたルール（メソッドの使用法）に従う事が原則となっています。
- 極端なベース・クラスの場合、ルールだけを定義する事が出来ます。この様なクラスをアブストラクト・クラスと言います。
- アブストラクト・クラスは未完成である為、インスタンスを作る事は出来ません。メソッドの定義は署名（signatures）だけから構成する様にします。その場合、修飾子pureを用います。

```
virtual class abstract_class;
pure virtual function print(string key,bit [31:0] data);
// ...
endclass
```

- サブクラスはprintの定義をしなければなりません。具体的な定義をしない場合、サブクラスもアブストクラスになり、インスタンスを作成する事は出来ません。
- UVMを理解するには、アブストラクト・クラスの概念を理解しなければなりません。

パラメータによる汎用クラスの定義（UVMを理解する為には必須の知識）

- クラスの定義にタイプ・パラメータを付加する事により、汎用的なクラスを定義する事が出来ます。C++、及び、Javaに同様の概念があります。
- 例えば、整数をキーにする辞書と文字列をキーにする辞書は同じ機能を持ちます。異なる点は、キーのデータ・タイプが異なるだけです。
- この様な場合、キーをパラメータKEY、データのデータ・タイプをパラメータDATATYPEとしてクラスを定義します。そして、クラス内では、KEYとDATATYPEを使用して論理を記述します。
- 使用する側は、クラスのインスタンスを作る時に、KEYとDATATYPEに実際のデータ・タイプを指定します。

パラメータによる汎用クラスの定義（その1）

```
 1  class map #(type KEY = int,type DATA = int);
 2  DATA    array[KEY];
 3  string  name;
 4
 5  function new(string name);
 6          this.name = name;
 7  endfunction
 8
 9  function void put(KEY key,DATA data);
10          array[key] = data;
11  endfunction
12
13  function DATA get(KEY key);
14          return array[key];
15  endfunction
16
17  function int exists(KEY key);
18          return array.exists(key);
19  endfunction
20  // ...
21  endclass
```

抽象的なタイプを使用して、汎用的な辞書を定義する。

タイプには標準値を定義して置くと親切、且つ、便利です。

パラメータによる汎用クラスの定義（その2）

- パラメータ化したクラスを使用する為には、具体的なタイプを指定する必要があります。

mapに対して具体的なタイプを指定する。

```
47  module test;
48  map #(string,real)      weight;
49  string                  key;
50
51          initial begin
52                  weight = new("Foods      Weight");
53
54                  weight.put("meat",1000.3);
55                  weight.put("bread",200.5);
56                  weight.put("rice",5000.9);
57
58                  $display("%s",weight.name);
59
60                  weight.print;
61
62                  weight.clear();
63                  weight.print();
64          end
65
66  endmodule
```

クロッキングブロック

クロッキング・ブロックは、クロックに同期して変化する信号のタイミング（skew）を設定する構文です。タイミングを設定すると、クロックのイベントに同期して、指定したタイミングで信号値が変化します。クロッキング・ブロックにより、個々の信号の変化を管理する必要は無くなります。

本章では、クロッキング・ブロックの基本機能を紹介します。

default clocking

- defaultを付加すると標準クロッキング（default clocking）となります。
- クロッキング・イベントが必要なSystemVerilog機能に於いてクロッキングの指定を省略すると、標準クロッキングが採用されます。
- 例えば、アサーションのsequenceやpropertyでdefault clockingを使用します。default clockingを定義すると、sequenceやpropertyの記述からクロッキング・イベントを省略する事が出来る為、より抽象度の高い記述をする事が出来ます。

```
 1  module test(input clk,req,ack);
 2
 3          default clocking cb @(posedge clk); endclocking
 4
 5          sequence check_req_ack;
 6                  req ##[1:3] ack;
 7          endsequence
 8
 9          assert property (check_req_ack)
10                  $display("@%0t: PASS",$time);
11                  else $display("@%0t: FAIL",$time);
12  // ...
13  endmodule
```

最も簡単なクロッキング・ブロック（重要な知識）

- 最も簡単なクロッキング・ブロックは以下の様になります。

```
clocking cb @(posedge clk); endclocking
```

- 実は、これだけでも意味があります。@(posedge clk)の代わりに、簡単に@(cb) と書く事が出来ます。タイピング負荷が軽減します。
- 更に、クロッキング・ブロック内のposedgeをnegedgeに変えるだけで@(cb) は @(negedge clk)に変わります。
- クロッキング・ブロック名称を随所で使用する為、一般に、クロッキング・ブロック名称を出来るだけ短く定義する事が勧められています。

インターフェース
(Interfaces)

SystemVerilogのインターフェースはモジュール間の接続を簡素化する為に作られました。機能的には、インターフェースはモジュール・ポートを一般化した概念ですが、実際の機能はそれ以上の効果を齎します。大規模なプロジェクトには必須の概念です。

UVMを使用する場合には、interfaceを理解しなければなりません。

インタフェース（interfaces）

- インターフェースは、モジュールと同じ様にインスタンスを作る事が出来ます。そして、インターフェースのインスタンスをポートして使用し、モジュール間の接続に使用します。
- インターフェースは複数の信号から構成されます。しかも、それらの信号の向き（input、inout、 output等）を参照するモジュール毎に変える事が出来ます。この機能はmodportにより実現されます。
- モジュール間の接続変更はインターフェース内の定義変更だけで済みます。それぞれのモジュールのポート・リストを変更する必要はありません。
- インターフェースにはinitial、及び、alwaysプロシージャを記述する事が出来ます。信号値の初期化、及び、プロトコルのチェック等を容易に行なう事が出来ます。

インタフェース（interfaces）（定義例）

```
 1  interface simple_if(input bit clk);
 2  logic [1:0]     grant, request;
 3  logic           reset;
 4
 5  clocking cb @(posedge clk);
 6  output  request;
 7  input   grant;
 8  endclocking
 9
10  modport TEST (output request, reset,input grant,clk);
11  modport DUT (input request,reset,clk,output grant);
12  modport MONITOR (input request,grant,reset,clk);
13
14  initial begin
15          grant <= 0;
16          request <= 0;
17          reset <= 0;
18  end
19
20  endinterface
```

インタフェース（interfaces）（使用例）

```
 1  module top;
 2  bit     clk;
 3
 4          initial forever #10 clk = ~clk;
 5
 6  simple_if INTF(clk);
 7  dut DUT(INTF);
 8  test TEST(INTF);
 9
10  endmodule
11
12  module test(simple_if intf);
13
14          initial begin
15                  @(posedge intf.clk) intf.request <= 2'b01;
16                  // ...
17          end
18
19  endmodule
```

インターフェースのインスタンスを作成する。

インターフェースをポートに指定している。

modport

- インターフェースには多くの信号が定義されていますが、一般に、一つのモジュールは一部の信号にしかアクセスしません。その様な場合、modportを使用して信号へのアクセスを制限します。例えば、次の様にmodportを使用します。

```
module test(simple_if.TEST intf);

        initial begin
                @(posedge intf.clk) intf.request <= 2'b01;
                // ...
        end

endmodule
```

- testでは、modport TESTに定義されているポート以外を見る事は出来ません。

virtual インターフェース（UVMで使用される重要な概念）

- virtualインターフェースはインターフェースのインスタンスを示す変数です。いわば、インターフェース・オブジェクトへのポインターです。このポインターをクラスのハンドルと同じ様に使用する事が出来ます。クラス内にはインターフェースを定義する事は出来ませんが、クラスはvirtualインターフェースを持つ事が出来ます。
- virtualインターフェースはnullで初期化される為、使用するまでにインターフェースのインスタンスで初期化しなければなりません。通常は、コンストラクタnewでvirtualインターフェースの初期化をします。例えば、以下の様にします。

```
class simple_driver;
virtual simple_if       vif;

function new(simple_if sif);
        vif = sif;
endfunction
// ...
endclass
```

- 但し、UVMを使用する場合には、virtualインターフェースをコンストラクタで初期化する方法は正しくありません。

パッケージ
(Packages)

パッケージは共有するリゾース（パラメータ、データ・タイプ、タスク、ファンクション等）のコンテイナーです。構文としては一番外側に位置します。即ち、他のスコープにパッケージを含む事は出来ません。

パッケージはライブラリー開発等で使用される重要な機能です。

パッケージ（packages）

- パッケージにはタスク、及び、ファンクションを含む事が出来ますが、initialやalways等のプロシージャを含む事は出来ません。
- パッケージを個別のファイルに定義した場合、他のファイルでパッケージを参照する為には`include文を使用します。例えば、次の様にファイルを引用します。

```
`include "my_package.sv"
```

- パッケージ内に定義されているリゾースを参照する為には、パッケージ名称にスコープ・オペレータ（::）を適用します。例えば、パッケージPkgに定義されているパラメータPIをモジュール内で参照する為には、

```
$display("PI=%g",Pkg::PI);
```

- の様にして参照します。スコープ・オペレータの使用を省略する為には、パッケージをimportしなければなりません。

パッケージのimport

- importする際、個別に名称を指定する事が出来ます。例えば、

```
module test;
import  Pkg::ARRAY_SIZE;
import  Pkg::setup, Pkg::print;
int     elm[];
// ...
```

- の様にしてimportする事が出来ます。importした名称は、モジュール test 内で Pkg:: を付けずに使用する事が出来ます。
- 個別にimportする代わりに、パッケージの全てのメンバーを同時にimportする為には　::* を使用します。例えば、次の様にして使用します。

```
module test;
import  Pkg::*;
int     elm[];
// ...
```

パッケージ（定義例）

```
1  `ifndef PKG_H
2  `define PKG_H
3
4  package Pkg;
5  parameter       ARRAY_SIZE = 16;
6  typedef enum { RED, YELLOW, GREEN } color_e;
7
8  function void setup(output int elm[]);
9          foreach(elm[i])
10                 elm[i] = i+1;
11 endfunction
12
13 function void print(string msg,int width,int elm[]);
14 string  format;
15         format = make_format(width);
16         $write("%s:",msg);
17         foreach(elm[i])
18                 $write(format,elm[i]);
19         $display;
20 endfunction
21
22 function string make_format(int width);
23         return $sformatf("%%%0dd",width);
24 endfunction
25 endpackage
26
27 `endif
```

パッケージ
（参照する例）

```
1  `include "Pkg.pkg"
2
3  module test;
4  import  Pkg::*;
5  int     elm[];
6
7          initial begin
8                  elm = new [ARRAY_SIZE];
9
10                 setup(elm);
11                 print("elm",3,elm);
12         end
13 endmodule
```

ランダム・スティムラスの生成
（Constrained Random Value Generation）

SystemVerilogには、効率よく、且つ、効果的にランダム・スティムラスを生成する機能があります。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出来ます。

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。この章で紹介する機能は実践で必要になります。

ランダム・スティムラス生成機能（Constrained Random Tests）

- 近年のテスト法は、CRT（Constrained Random Tests）をベースしています。CRTは、制約を満たすテスト・データをランダムに生成します。
- CRTは、膨大な検証空間を効率よく処理する為の効果的な方法です。従来の手作業によるデータ生成（directed tests）は、CRTがカバー出来ない特殊な集合を検査する為に使用されます。
- SystemVerilogは、直感的で、使い易いCRT機能を備えています。乱数を発生すべきデータ項目にrand、又は、randc修飾子を添えるだけで乱数を発生する事が出来ます。また、制約は、通常の式を用いて設定する事が出来ます。

ランダム・スティムラスの生成（その1）

- 生成の仕方は直接的で明確です。テスト・データに乱数発生の情報を付加するだけで準備が終了します。後は、乱数を発生すべきタイミングを選び乱数を発生させます。例えば、次の様にしてパケットを準備します。

```
class packet_t;
        rand bit [15:0] addr;
        rand bit [31:0] data;
        int unsigned    state;
endclass
```

- クラスには乱数を発生する関数randomize()が定義されているので、次の様にして乱数を発生する事が出来ます。

```
packet_t        packet;
packet = new;
// ...
assert( packet.randomize() );
```

- 関数randomize()はaddr及びdataに自動的に乱数を割り当てます。一方、stateには乱数発生情報が付加されていない為、乱数は割り当てられません。

制約によるランダム・スティムラスの生成（その2）

- 前例では制約を付けていない為、意味のないテスト・データが生成される可能性があります。そこで、次の様にして制約を追加する事が出来ます。

```
class packet_t;
        rand bit [15:0] addr;
        rand bit [31:0] data;
        int unsigned    state;
        constraint boundary_condition { addr[1:0] == 2'b00; }
endclass
```

- この制約によりaddrは常に4の倍数となる様に乱数が割り当てられます。
- dataは32ビットなので、制約なしで乱数を発生させても余り意味がありません。然し、どの様な制約を定義するかはテスト・ケースに依存する可能性があります。この様な状況に於いては、クラスの特性を利用する方法が得策です。即ち、クラスの継承をして制約を追加します。

制約によるランダム・スティムラスの生成（その3）

- クラスの継承は以下の様になります。

```
class small_packet_t extends packet_t;
        constraint small_data { data inside { [0:128], [512:1024] }; }
endclass
```

- クラスの継承なので、small_packet_tには制約boundary_conditionも有効になっています。実は、もう少し一般的にする事が出来ます。small以外に、medium、及び、largeに備える事で応用が広くなります。例えば、次の様にします。

```
typedef enum { SMALL, MEDIUM, LARGE } test_size_e;

class simple_packet_t extends packet_t;
        rand test_size_e        test_size;
        constraint addr_selection {
                solve test_size before data;
                test_size == SMALL -> data inside { [0:32] };
                test_size == MEDIUM -> data inside { [30000:32000] };
                test_size == LARGE -> data inside { [100000:200000] };
        }
endclass
```

ランダムスティムラス生成機能（使用例）

```
1  class Packet;
2          rand bit [31:0] src, dst, data[3];
3          randc bit [1:0] kind;
4          constraint C
5          {
6                  src > 10; src < 15;
7                  dst != src && dst < 24 && dst > 5;
8                  foreach(data[i])
9                          data[i] < 128 && data[i] > 64;
10         }
11 endclass
12
13 module test;
14 Packet  p;
15 int     max_tries;
16         initial begin
17                 p = new;
18                 max_tries = 16;
19                 while( max_tries-- > 0 ) begin
20                         #10 assert (p.randomize())
21                                 else $fatal(0,"Packet::randomze failed");
22                         printPacket(p);
23                 end
24         end
25 // ...
26 endmodule
```

UVM概要
(Universal Verification Methodology)

この章では、UVMとは何かを説明します。本章により、UVMの構造が自然と明らかになります。

UVMを使用する為の技術は次の章で解説します。

UVMとは何か？

- UVM (Universal Verification Methodology)とは、検証分野で推奨されている技術、ルール、慣習、規律等をコードとして具体化し、検証技術の再利用性と生産性向上をさせる為のSystemVerilogのクラス・ライブラリーです。
- UVMはAccellera Systems Initiativeにより開発されました。
- UVMはSystemVerilogをベースにして記述されているので、SystemVerilogをサポートしている検証ツールの環境で使用する事が出来ます。
- UVMは、検証分野で推奨されている技術、ルール、慣習、規律等をベースにしている方法論である為、その開発の背景を最初に理解しなければなりません。

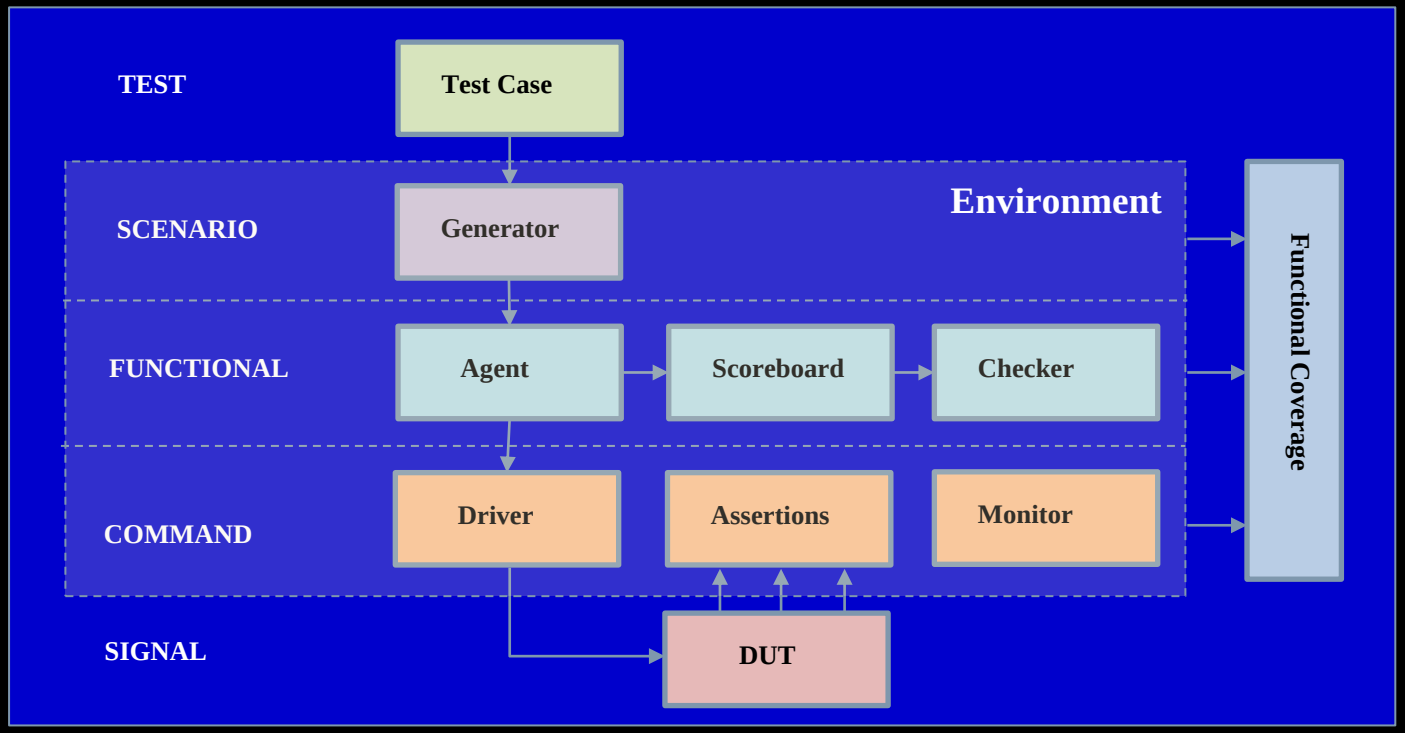
検証技術のトレンド（VMM）

- 近年の検証技術では階層的にテストベンチを記述する手法（layered testbench）が採用されています（[3]）。

Layer	検証目的及び機能
test layer	テストベンチのトップ・レベルのレイヤーです。テストを生成して下位の層に送ります。
scenario layer	シナリオに沿ってトランザクションのシーケンスを生成します。
functional layer	高位のレベルのコマンドを処理するレイヤーです。例えば、DMA read/write等のコマンドを受け取ると、個々のbus read/write等のコマンドに分解してcommand layerに送ります。
command layer	トランザクション・レベルのコマンドをシグナル・レベルの値に変換をしてsignal layerに送ります。DriverはDUTをドライブします。
signal layer（DUT）	DUTからのレスポンスは上位の層に送られます。アサーション等の結果も上位の層に戻されます。

テストベンチとレイヤーの関係（[3]）

- このlayered testbenchを良く考察するとUVM のメソドロジー・クラスの名称に対応している事が分かります。言い換えると、UVMは検証技術のトレンドに沿って開発されている事を確認する事が出来ます。

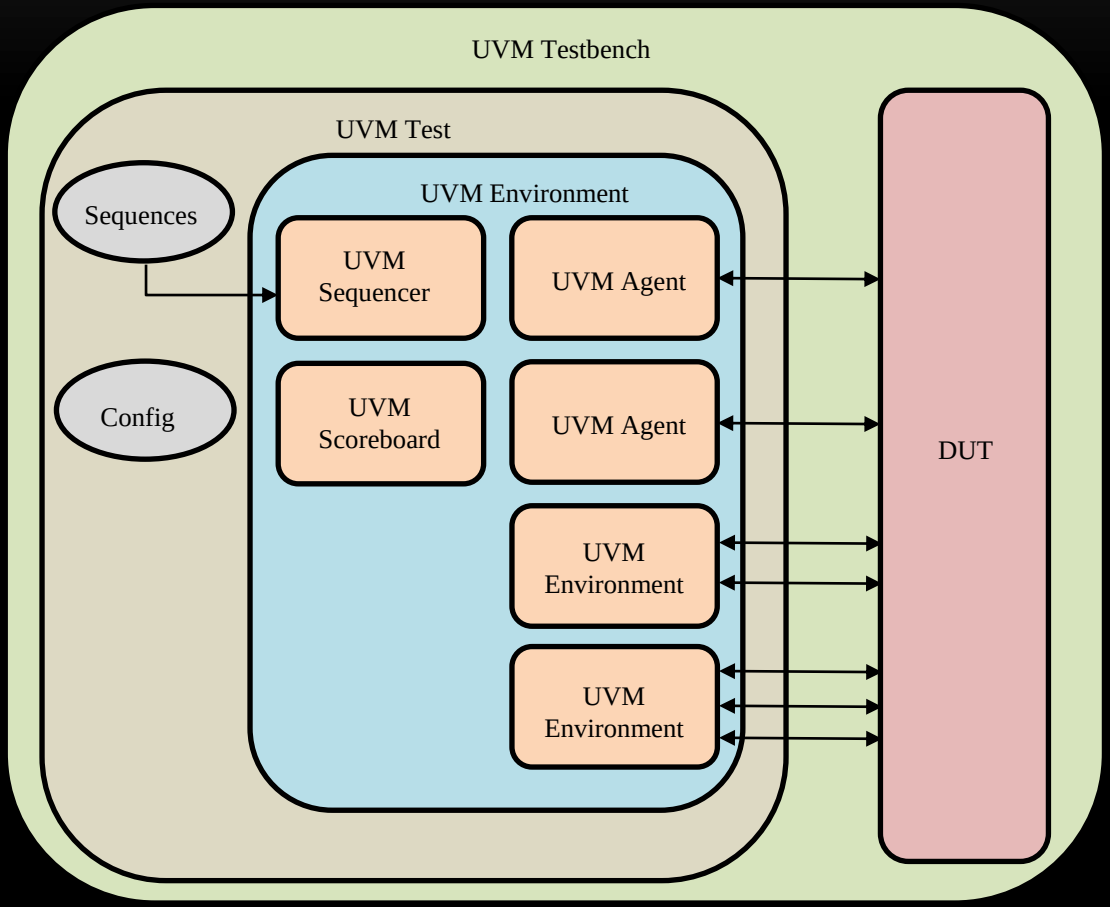


代表的なUVMクラス

- UVMには多くのクラスが定義されていますが、ユーザが直接使用するのはその内の一部のクラスで、メソドロジー・クラス（methodology class）と呼ばれます。

UVMクラス	種別
uvm_sequence_item	トランザクション関連
uvm_sequence	トランザクション関連
uvm_driver	メソドロジー・クラス
uvm_sequencer	メソドロジー・クラス
uvm_env	メソドロジー・クラス
uvm_agent	メソドロジー・クラス
uvm_test	メソドロジー・クラス
uvm_monitor	メソドロジー・クラス
uvm_scoreboard	メソドロジー・クラス

UVMテストベンチの構造（[2]）



TLM (Transaction Level Modeling)

- UVMはTLMを採用し、シグナル・レベルよりも高位の記述法を用いて検証タスクを表現します。このアプローチはシステムの動作を考察する際の自然な方法です。
- トランザクションは二つのコンポーネント間の通信をモデルする為に必要な情報を意味します。トランザクションには、その情報を操作する為のメソッドが定義されます。
- UVMではトランザクションはオブジェクトであり、UVMコンポーネントがトランザクションを操作します。その内の特別なコンポーネントとしてドライバー（uvm_driverのサブクラス）が存在します。ドライバーはトランザクションをシグナル・レベルに変換してDUTを操作する役目を持ちます。
- DUT側のシグナルの変化を検知する役目を持つUVMコンポーネントも必要になります。そのコンポーネントは、一般的には、コレクター（collector）と呼ばれます。
- ドライバーとコレクターの存在により、UVMではトランザクション・レベルでシステムを記述する事が出来る様になります。尚、UVMコンポーネントとDUT間のデータ授受にはSystemVerilogのvirtual interfaceが使用されます。

uvm_object

- UVMはSystemVerilogクラスとUVMマクロから構成されます。UVMには多くのクラスが定義されていますが、特別なクラスとしてuvm_objectが存在します。このクラスはアブストラクト・クラスで他の全てのUVMクラスのベース・クラスになっています。
- uvm_objectクラスでは他の全てのクラスに共通する属性、及び、手順を宣言しています。具体的な手順の内容はサブクラスで行います。手順としては、例えば、print及びcopyがあります。
- uvm_objectクラスから二つの重要なサブクラスが定義されています。それらは、uvm_sequence_itemとuvm_componentです。前者は、トランザクションを定義する為に使用します。一方、後者はテストベンチを構築する為に使用します。全てのメソドロジー・クラスはuvm_componentのサブクラスです。

UVMコンポーネント（uvm_componentのサブクラス）

- UVMコンポーネントはシミュレーションの対象となるので、デザインにおけるmoduleの様な役目を果たします。即ち、UVMにおいて、コンポーネントは自然にコンポーネント・インスタンスの階層構造を構成します。
- 階層のトップには、uvm_topと呼ばれるインスタンスが存在します。UVMはその階層構造を使用してダイナミックなシミュレーションを制御します。例えば、階層構造を使用して、テストベンチの構成をシミュレーション開始直前に変える事が出来ます。この機能により、一度のコンパイルで複数のテスト・ケースを実行する事が出来ます。
- 階層構造は、シミュレーション開始直前に決定し、一度シミュレーションが開始（コンポーネントのタスクrun_phase() の実行を開始した時点）するとコンポーネント・インスタンスの階層構造は変化しません。

UVMを使用する為に必要な手順

```
1  `include "uvm_pkg.sv"
2  `include "uvm_macros.svh"
3
4  module top;
5  import uvm_pkg::*;
6
7  class my_comp extends uvm_component;
8          int     data[];
9          int     data_size = 2;
10         string  header_name = "data[0..1]";
11
12         // ...
13 endclass
14
15 my_comp         test;
16
17         initial begin
18                 // fix the configuration.
19                 uvm_config_db#(string)::set(null,"test","header_name","data[0..7]");
20                 uvm_config_db#(int)::set(null,"test","data_size",8);
21                 // create components.
22                 test = my_comp::type_id::create("test",null);
23                 run_test();
24         end
25
26 endmodule
```

① UVMクラス・ライブラリーを`includeする。

② uvm_pkgをimportする。

③ テストすべき内容を定義する。

④ run_test()を呼び出しテストを実行する。

重要な概念：run_test()

- ユーザが準備したトップ・モジュールからrun_test()を呼び出すと、UVMが実行権を得て、シミュレーション終了時まで実行制御権を持ち続けます。
- メソドロジー・クラスを使用してユーザが記述したUVMコンポーネントはUVMにより呼び出し制御を受けます。ユーザ側に実行制御権はありません。
- ユーザのUVMコンポーネントに制御が渡るタイミングは予め決定されていて、シミュレーション・フェーズ（simulation phases）と呼ばれています。

シミュレーション・フェーズ（virtual function又はtask）

フェーズ	機能
build_phase	コンポーネントの階層を構築するフェーズです。従って、階層のトップから順に呼ばれて行きます。通常、チャイルド・コンポーネントをこのフェーズで作成します。
connect_phase	コンポーネント間の接続を完成するフェーズです。例えば、TLMポートの接続を定義します。
end_of_elaboration_phase	全ての接続が終了するとこのフェーズに制御が移ります。通常は、コンフィギュレーションをプリントする等の処理を記述します。
start_of_simulation_phase	シミュレーションが開始する直前にこのフェーズが呼ばれます。初期化処理等を行う事が出来ます。
run_phase	シミュレーションを行う為のフェーズです。
extract_phase	シミュレーションが終了すると、このフェーズに制御が移ります。シミュレーション結果を抽出する為の処理を記述する事が出来ます。
check_phase	抽出したシミュレーション結果をチェックする為の処理を記述します。
report_phase	シミュレーション結果のレポートを出力する処理を記述します。

シミュレーション・フェーズの記述例

```
 1  //
 2  //      my_testbench
 3  //
 4  class my_testbench extends uvm_test;
 5  `uvm_component_utils(my_testbench)
 6  //      new
 7  function new(string name="my_testbench",uvm_component parent=null);
 8          super.new(name,parent);
 9  endfunction
10  //      build_phase
11  function void build_phase(uvm_phase phase);
12          super.build_phase(phase);
13          `uvm_info("MY_TESTBENCH","build_phase running.",UVM_LOW)
14  endfunction
15  //      run_phase
16  task run_phase(uvm_phase phase);
17          `uvm_info("MY_TESTBENCH","run_phase running.",UVM_LOW)
18  endtask
19  endclass
20
```

データ・オブジェクトとフィールド・マクロ

- データ・オブジェクトを定義する場合には、プロパーティ名に対して`uvm_object_utilsマクロを使用しなければなりません。

```
21  class simple_item extends uvm_sequence_item;
22  rand int unsigned addr;
23  rand int unsigned data;
24  rand int unsigned delay;
25  rand simple_item_delay_e delay_kind;
26  `uvm_object_utils_begin(simple_item)
27          `uvm_field_int(addr,UVM_DEFAULT)
28          `uvm_field_int(data,UVM_DEFAULT)
29          `uvm_field_int(delay,UVM_DEFAULT)
30          `uvm_field_enum(simple_item_delay_e,delay_kind,
31                  UVM_DEFAULT|UVM_NOCOMPARE|UVM_NOPACK)
32  `uvm_object_utils_end
33
34  //
35  //      new
36  //
37  function new(string name="simple_item");
38          super.new(name);
39  endfunction
40  endclass
```

コンポーネントとフィールド・マクロ

- コンポーネントに関しては、`uvm_component_utilsマクロを使用しなければなりません。

```
 8  class my_comp extends uvm_component;
 9  int      data[];
10  int      data_size = 2;
11  string   header_name = "data[0..1]";
12
13  `uvm_component_utils_begin(my_comp)
14          `uvm_field_string(header_name,UVM_DEFAULT)
15          `uvm_field_int(data_size,UVM_DEFAULT)
16          `uvm_field_array_int(data,UVM_DEFAULT)
17  `uvm_component_utils_end
18
19  function new(string name,uvm_component parent);
20          super.new(name,parent);
21  endfunction
22
23  // ...
24
25  endclass
```

raise_objection()とdrop_objection()（UVMで最も難解な概念の一つ）

- UVMコンポーネントを正しく記述してもシミュレーションは実行しません。厳密に言えば、シミュレーションは時刻0で終了してしまいます。
- シミュレーションをする為には、UVMに実行すべき内容がある事を知らせなければなりません。そして、実行が終了したらその旨通知しなければなりません。
- raise_objection() はUVMに実行すべき内容がある事を知らせます。一方、drop_objection() は実行すべき内容が完了した事をUVMに知らせます。全ての実行すべき内容が完了した時点でUVMはシミュレーションphaseを終了して、次のphase（通常は、extract_phase）に進みます。
- テストベンチを構成する少なくとも一つのコンポーネントが、run_phase()の初めにraise_objection()を呼び、処理終了後にdrop_objection()を呼ばなければなりません。

raise_objection()とdrop_objection()の使用例

- 実行開始前にraise_objection()を呼び、実行を終了した時点で、drop_objection()を呼び出します。

```
 1  class env extends uvm_env;
 2
 3    local pin_vif pif;
 4    driver d;
 5
 6    function new(string name, uvm_component parent = null);
 7      super.new(name, parent);
 8      d = new("driver", this);
 9    endfunction
10
11    task run_phase(uvm_phase phase);
12      phase.raise_objection(this);
13      do_something();
14      phase.drop_objection(this);
15    endtask
16
17  // ...
18
19  endclass
```

UVMの検証への適用
(Verification Using UVM)

本章では、UVMを使用する際に必要となる知識、及び、ガイドラインを解説します。実践で必ず役に立つ知識を纏めました。

コンストラクタ（データ・オブジェクト）

- UVMでは、コンストラクタに対して一般的なルールがあります。データ・オブジェクトでは、名称に対して標準値を設定します。例えば、次の様に名称の標準値を定義します。更に、super.new()を呼び出す事が必要です。

```
21  class simple_item extends uvm_sequence_item;
22  rand int unsigned addr;
23  rand int unsigned data;
24  rand int unsigned delay;
25  rand simple_item_delay_e delay_kind;
26  `uvm_object_utils_begin(simple_item)
27          `uvm_field_int(addr,UVM_DEFAULT)
28          `uvm_field_int(data,UVM_DEFAULT)
29          `uvm_field_int(delay,UVM_DEFAULT)
30          `uvm_field_enum(simple_item_delay_e,delay_kind,
31                  UVM_DEFAULT|UVM_NOCOMPARE|UVM_NOPACK)
32  `uvm_object_utils_end
33
34  //                          標準値を定義して置く。
35  //      new                 ↓
36  //
37  function new(string name="simple_item");
38          super.new(name);    ← super.new()を呼ぶ。
39  endfunction
40  // ...
41  endclass
```

コンストラクタ（コンポーネント）

- コンポーネントは階層を構築する為、親コンポーネントの指定が必要になります。また、インスタンス名称には標準値を設定しません。同じ階層レベル内には、ユニークなインスタンス名称が必要な為、この様な配慮が必要になります。例えば、次の様に定義します。更に、super.new()を呼び出す事が必要です。

```
 8  class my_comp extends uvm_component;
 9  int      data[];
10  int      data_size = 2;
11  string   header_name = "data[0..1]";
12
13  `uvm_component_utils_begin(my_comp)
14          `uvm_field_string(header_name,UVM_DEFAULT)
15          `uvm_field_int(data_size,UVM_DEFAULT)
16          `uvm_field_array_int(data,UVM_DEFAULT)
17  `uvm_component_utils_end
18
19  function new(string name,uvm_component parent);
20          super.new(name,parent);
21  endfunction
22  //...
23  endclass
```

標準値を定義しない。

親コンポーネントを指定。

ファクトリ
（絶対に遵守すべきルール）

- オブジェクト、及び、コンポーネントのインスタンスを作る場合、newオペレータを使用する代わりに、factoryメソッドを使用する事が推奨されています。

```
37  my_comp          test;
38
39  initial begin
40          // fix the configuration.
41          uvm_config_db#(string)::set(null,"test","header_name","data[0..7]");
42          uvm_config_db#(int)::set(null,"test","data_size",8);
43          // create components.
44          test = my_comp::type_id::create("test",null);
45          run_test();
46  end
```

newオペレータを使用せずにファクトリ・メソッドを使用してインスタンスを作成する。

サブコンポーネントの生成（絶対に遵守すべきルール）

- 通常、build_phase()でサブコンポーネントの生成をします。コンストラクタが実行する時点ではコンフィギュレーションが決定されていない為、コンストラクタ内でサブコンポーネントを作成する事は正しくありません。

```
49  class simple_master_agent extends uvm_agent;
50  protected int master_id;
51  simple_driver           driver;
52  simple_sequencer        sequencer;
53  simple_collector        collector;
54  simple_monitor          monitor;
55  `uvm_component_utils_begin(simple_master_agent)
56          `uvm_field_int(master_id, UVM_DEFAULT)
57          `uvm_field_enum(uvm_active_passive_enum,is_active,UVM_DEFAULT)
58  `uvm_component_utils_end
59  // ...
60  function void build_hase(uvm_phase phase);
61          collector = simple_collector::type_id::create("simple_collector",this);
62          monitor = simple_monitor::type_id::create("simple_monitor",this);
63          if( is_active == UVM_ACTIVE ) begin
64                  sequencer = simple_sequencer::type_id::create("simple_sequencer",this);
65                  driver = simple_driver::type_id::create("simple_driver",this);
66          end
67  endfunction
68  // ...
69  endclass
```

サブコンポーネント

サブコンポーネントとフィールド・マクロ

- サブ・コンポーネントにはフィールド・マクロを指定する必要はありません。寧ろ、指定してはいけません。
- uvm_agentのサブクラスの場合、is_activeに対してフィールド・マクロを定義しなければならない。

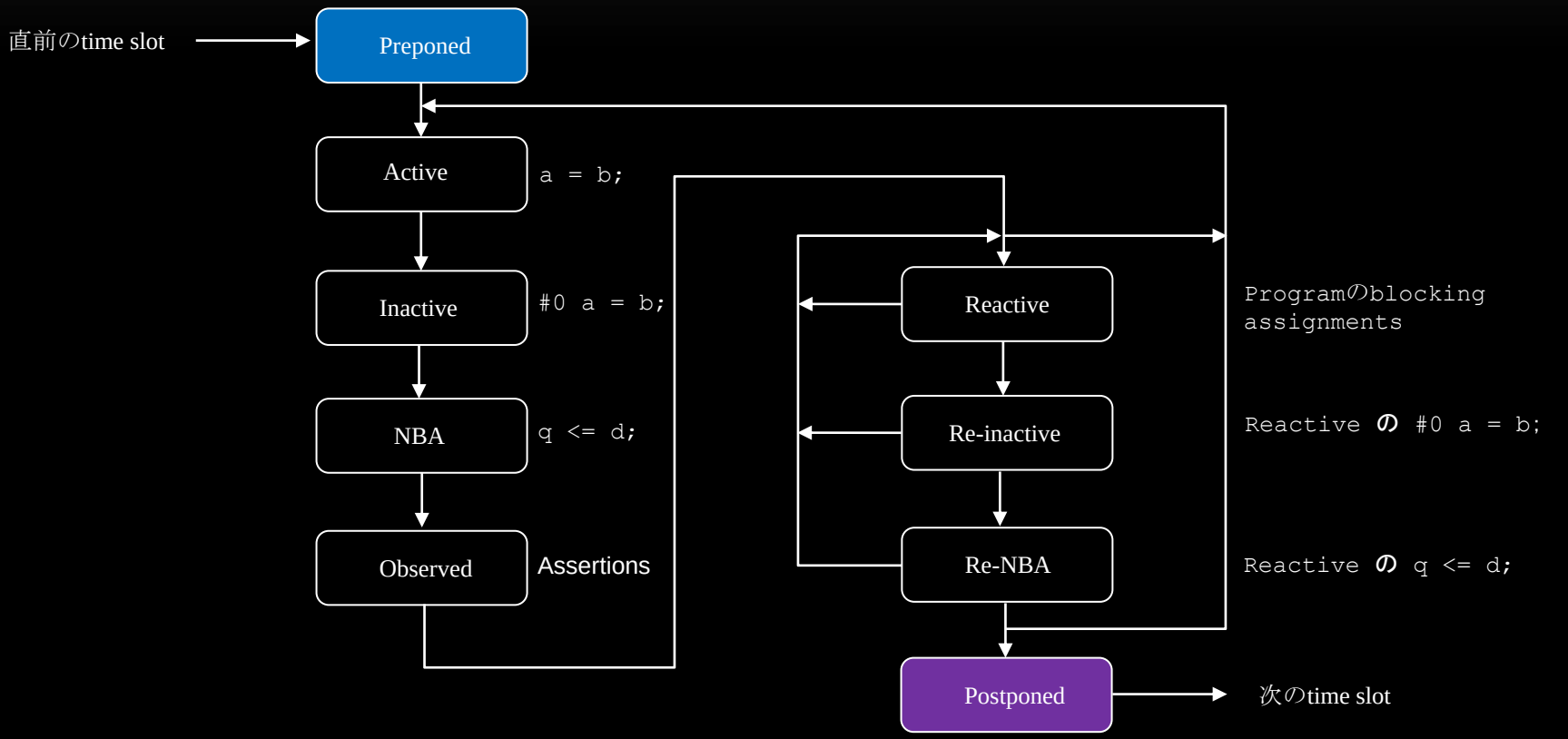
```
49  class simple_master_agent extends uvm_agent;
50  protected int master_id;
51  simple_driver           driver;
52  simple_sequencer        sequencer;
53  simple_collector        collector;
54  simple_monitor          monitor;
55
56  `uvm_component_utils_begin(simple_master_agent)
57          `uvm_field_int(master_id, UVM_DEFAULT)
58          `uvm_field_enum(uvm_active_passive_enum,is_active,UVM_DEFAULT)
59  `uvm_component_utils_end
60
61  function new(string name,uvm_phase phase);
62          super.new(name,phase);
63  endfunction
```

サブコンポーネント

is_activeのフィールド・マクロが必要。

付録

シミュレーション領域（[1]）（simulation regions）



参考の為に、各regionには代表的な実行命令が添えられています。
完全な、regions図は文献[1]を参照して下さい。

参考文献

本資料公開後、多くの部分が読み易い書物として編纂されています。興味のある方は文献[9-11]を参照ください。

[1] IEEE Std 1800-2017: IEEE Standard for SystemVerilog – Unified Hardware Design, Specification and Verification Language.

[2] Universal Verification Methodology (UVM) 1.2 User's Guide, Accellera, October 8, 2015.

[3] Chris Spear: SystemVerilog for Verification, 2nd Edition, Springer 2008.

[4] Kathleen A. Meade and Sharon Rosenberg: A Practical Guide to Adopting the Universal Verification Methodology (UVM), 2nd Edition Cadence Design Systems, Inc. 2013.

[5] Ashok B. Mehta: SystemVerilog Assertions and Functional Coverage, Springer 2014.

[6] Ben Cohen, Srinivasan Venkataramanan, Ajeetha Kumari, and Lisa Piper: SystemVerilog Assertions Handbook, 4th Edition, VhdlCohen Publishing, 2016.

[7] UVM 1.2 Class Reference, Accellera.

[8] Stuart Sutherland, Simon Davidmann, and Peter Flake: SystemVerilog for Design, 2nd Edition, Springer 2006.

[9] 篠塚一也、SystemVerilogによる検証の基礎、森北出版 2020.

[10] 篠塚一也、SystemVerilog入門、共立出版 2020.

[11] 篠塚一也、実践UVM入門、森北出版 2021.

✓SystemVerilog