Artgraphics

SystemVerilog 雑談

Document Revision: 2.4, 2022.06.12

アートグラフィックス
篠塚一也

SystemVerilog 雑談

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

SystemVerilog Monologue

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

はじめに

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

本資料には、筆者が気づいた SystemVerilog の話題が無作為に選び出されて解説されています。過去にツイートした内容でも、強調する価値があると思われる題材は、一部を書き直して、本資料に再度掲載されていますので、参考にして下さい。尚、題材がツイートに対応している場合には、日付が添えてあります。適時に情報を得たい方は、Twitter で SystemVerilog を検索すると良いです。

本資料の内容は完成形ではなく、随時更新されます。筆者が思いついた時に本資料を編集していくため、いつ完成するか分かりません。本資料の内容が更新された場合には Twitter で告知する予定なので、適宜、最新版をダウンロードして下さい。

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。

最後に、本資料を好きな時に、しかも気楽に読んで下さい。

アートグラフィックス
篠塚一也

1　マルチプレクサと論理合成　2020.09.20

合成結果が予測可能ならば、論理合成 をブラックボックスとして扱うのは妥当です。そうでない場合には、RTL コードの吟味が必要です。例えば、以下の SystemVerilog 記述が 2:1 の multiplexer を生成しても不思議ではありません。

```
module df(input [1:0] d,logic s,output out);
assign out = d[s];
endmodule
```

参考 1-1

実際、s==0 であれば d[0]が、s==1 であれば d[1]が、それぞれ out に設定されるので、マルチプレクサの動作を表現しています。

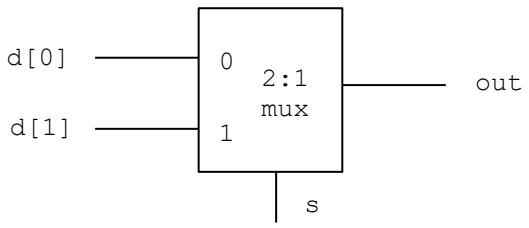


図 1-1　packed アレイによるマルチプレクサの合成

□

2　レベルセンシティブとエッジセンシティブ　2020.09.26

周知のように、レベルセンシティブとエッジセンシティブの制御は全く異なります。SystemVerilog では、両者の相互変換が可能です。文献[1,8,9]にも詳しい解説があります。

表 2-1　レベルセンシティブとエッジセンシティブの相互変換

レベルセンシティブ	エッジセンシティブ	説明
`wait( ev.triggered )`	`@ev`	`ev` には値が残らないので、`@ev` はイベント解除を見逃しやすい。一方、`ev.triggered` はレベルセンシティブなので、イベント解除を見逃さない。
`clear`	`$rose(clear)` `$fell(clear)`	レベルセンシティブであると、無駄に制御が発生する場合が多い。特に、アサーションの antecedent の場合にはエッジセンシティブな記述が必要。

3　タイミング制御と信号値の更新　2021.06.22

さすがの SystemVerilog でも正解が一意的に定まらない場合があるようです。

```
assign p = q;

initial begin
    q = 1;
    #1 q = 0;
    $display(p);
end
```

この場合、$display システムタスクが 0 または 1 の何れをプリントしても正しいと LRM では言っています。

q に 0 をセットした後、そのまま続行すれば、1 がプリントされ、続行する前に p の値を更新すれば 0 がプリントされます。

4 always @(*) 2021.07.19

SystemVerilog の@* は記述に使用されている名称からセンシティビティリストを生成しますが、例外条件を持ちます。その条件さえ理解していれば、@* は安全です。

@(*)は、記述で使用されているネットと変数からセンシティビティリストを生成しますが、以下の名称を除外します。

- `wait` またはイベント式のみに使用されている名称
- LHS のみに指定されている名称
- サブルーティン内部で使用されている名称

表 4-1 @(*)によるセンシティビティリスト生成例

記述	生成されるセンシティビティリスト
always @(*) y = (a & b) | (c & d) | myfunction(f);	@(a or b or c or d or f)

always @(*) begin
tmp1 = a & b;
tmp2 = c & d;
y = tmp1 | tmp2;
end
	@(a or b or c or d or tmp1 or tmp2)

always @(*) begin @(i) kid = b; end	@(b)

always @(*) begin
y = 8'hff;
y[a] = !en;
end
	@(a or en)

参考 4-1

関連する機能としては、`always_comb` がありますが、この機能は制約が強いため使用できない場合もあります。例えば、LHS がマルチドライバーを持つ場合には、`always_comb` を使用できません。したがって、`always @(*)`を理解しておくと便利な時もあります。
□

5　posedge と negedge　2021.08.05

SystemVerilog のエッジセンシティブイベントは使用されている式の値の LSB で判断されますが、イベントが発生したか否かの判定は、エッジが上を向くか下を向くかで判定されます。然し、シミュレータ開発者にとっては、結構面倒な判定になります。

LRM 217 ページより

SystemVerilog の posedge と negedge の判定

From	To			
	0	1	x	z
0	no edge	posedge	posedge	posedge
1	negedge	no edge	negedge	negedge
x	negedge	posedge	no edge	no edge
z	negedge	posedge	no edge	no edge

要約すると、

posedge	0 から他の値への変化。あるいは x、または z から 1 への変化。
negedge	1 から他の値への変化。あるいは x、または z から 0 への変化。

6 always_comb 2021.08.15

SystemVerilog の `always_comb` は、組み合わせ回路を作るための魔法ではありません。寧ろ、おまじないです。

簡単に言えば、`always_comb` は以下の機能を持ちます。詳細は、LRM 207 ページを参照ください。

- センシティビティリストを自動生成する。
- プロシージャ内で LHS として使用された変数は、他のプロシージャで LHS として使用する事はできません（シングルドライバーの制約）。
- プロシージャ内ではイベント制御、タイミング制御等の機能を使用する事はできません。
- 記述した回路が組み合わせ回路の条件を満たさなければ、警告を発行します。
- プロシージャは、`$time==0` の時、自動的に一度実行します。この機能は、シミュレーションのためであり、論理合成とは関係がありません。

つまり、論理合成の観点からは、`always_comb` はチェック機能が強化された `always` プロシージャと言えます。

参考 6-1

組み合わせ回路を記述する時、`always_comb` は便利ですが、その機能を理解したうえで使用する事が必要です。
□

7　論理合成と Shannon の展開定理　2021.08.26

Shannon の展開定理を参考にすると、SystemVerilog 記述の論理合成結果を予測できるようになります。この方法は、論理合成を知る契機になると思います。

n 変数のブール式に関する Shannon の展開定理（Boole の展開定理とも呼ばれる）とは、以下の等式を意味します。この定理は、(n-1)変数の式を ite で結合して、n 変数の式を表現します。ite は if-then-else の略で、MUX2 回路に相当します。

$$f(x_1, x_{2,} \dots, x_n) = ite(x_i, f_{x_i}, f_{x_i'}) = x_i * f_{x_i} + x_i' * f_{x_i'}$$

この定理は、divide-and-conquer メソッドの典型的なアルゴリズムです。論理合成結果を予測する簡単な例を紹介します。以下は、厳密な理論ではなく、大雑把な解説です。

表 7-1　Shannon の展開定理の合成結果予測への応用例

$f(s,a,b)$	Shannon の展開定理 (Boolean expression)			回路
	f_s	$f_{s'}$	展開	

always @(a,b,s)
if(s == 0)
out = a;
else
out = b;
	b	a	$s*b+s'*a$	MUX2

always @(b,s)
if(s == 0)
out = 0;
else
out = b;
	b	0	$s*b+s'*0 \rightarrow s*b$	AND

always @(a,s)
if(s == 0)
out = a;
else
out = 1;
	1	a	$s*1+s'*a \rightarrow s+s'*a \rightarrow s+a$	OR

- 合成可能な組み合わせ回路になる記述に対して、この定理を適用できます。
- 合成結果を最適化する必要があります。
- 上記の例は 1 ビット変数ですが、N ビットでも同様です。
- case 文も同様です。
- 詳しい解説は、「SystemVerilog アラカルト」にもあります。

8　always_comb とタスク呼び出し　2021.09.01

always_comb の内部にファンクションの呼び出しを使用できるのは明らかですが、タイミング制御を伴わないタスクであれば、always_comb の内部で使用できます。ただし、以下のような制限が付きます（LRM 207 ページ）。

タスク呼び出しが always_comb の内部で行われた場合、タスクの内部で使用されている信号は、センシティビティリストに反映されません。

ただし、使用するツールがその制限を取り除いている可能性はあります。しかし、一般には、always_comb 内ではタスク呼び出しを避ける事を勧めます。

参考 8-1

上記の制限があるため、SystemVerilog LRM では、タイミング制御を持たないタスク定義を void function に変換しても良いと規定しています。ツールがそのような変換を自動的に行えば、タイミング制御を持たないタスクを呼び出しても、上記の制限は取り除かれます。使用しているツールが、その変換機能を備えているかどうかは、マニュアルを参照下さい。
□

9 std::process 2021.09.11

SystemVerilog の std::process には、ガーベッジコレクション機能が備わっています。

- initial や always プロシージャは、process クラスのオブジェクトとして実行します。fork で生成されるプロセスも process クラスのオブジェクトとして実行します。
- process オブジェクトは、ユニークであり、検索用のキーとして使用する事ができます (LRM 230)。
- プロセスが終了して、どこからも参照されなくなると、process オブジェクトは再利用されるように管理されます (LRM 230)。したがって、多くのプロセスを生成しても、正常にプロセスを終了させれば、資源を有効に活用できます。そのためには、process クラスに定義されているメソッドを使用する必要があります。
- process は、クラスですが、semaphore や mailbox と異なり、ユーザが拡張する事はできません (LRM 230)。

参考 9-1

もし process クラスがユーザにより拡張可能であれば、拡張したクラスのコンストラクタを使用してプロセスを自由自在に作る事ができるようになります。これは、明らかに矛盾をします。したがって、process クラスをユーザは拡張する事はできません。
□

10　プロシージャ　2021.09.25

SystemVerilog の initial や always プロシージャは、独立したプロセスとして生成され、並列に実行しますが、final プロシージャは、独立したプロセスとして実行されずに、シミュレータから function のように呼び出されます。

表 10-1　SystemVerilog のプロシージャに関する規定

機能	規定	LRM のページ
initial と always	initial プロシージャと always プロシージャの実行順序に関するルールはありません。つまり、initial プロシージャが always プロシージャよりも先にスケジューリングされて実行するとは限りません。したがって、テストベンチが DUT よりも先に実行を開始する保証はありません。	205
initial	initial プロシージャは、一回だけ実行し、最後の命令が終了すると、プロシージャの終了となります。	205
always	繰り返し実行し、シミュレーションが終了する時のみ、always プロシーシャが終了します。	205
final	• シミュレーションの終了が宣言されると、final プロシーシャは一度だけ実行します。 • final プロシーシャに記述できる命令は、function 内で使用できる命令に限られます。 • initial や always プロシーシャと異なり、final プロシーシャは個別のプロセスとして実行しません。単なる function のようにシミュレータから呼び出され、シミュレーション時間を消費せずに戻ります。 • 複数の final プロシーシャを定義できますが、一定の順序で実行されます。順序はランダムではなく、deterministic です。つまり、ツールはルールに基づき実行順序を決定します。 • 全ての final プロシーシャが終了するとシミュレーションが終了します。 • final プロシーシャ内で、$finish を実行すると、即刻シミュレーションが終了します。	205 209

11　ポートの種類と接続　2021.10.02

SystemVerilog では、ポートの方向でポートの種類がほぼ決定されます。そして、ポートの種類は、接続法を規定します。

表 11-1　ポートの方向とポートの種類

ポートの方向	ポートの種類	接続可能なタイプ
input	ネットまたは変数	式を接続できます。input ポートは読み込み専用なので、ポートの種類がネットでも変数でも大きな違いはありません。
inout	ネット	ネットまたはネットの concatenation しか接続できません。
output	ネットまたは変数	output ポートは、ネットおよび変数に接続できます。それらの concatenation も接続可能です。
ref	変数	同等なデータタイプを持つ変数に接続できます。

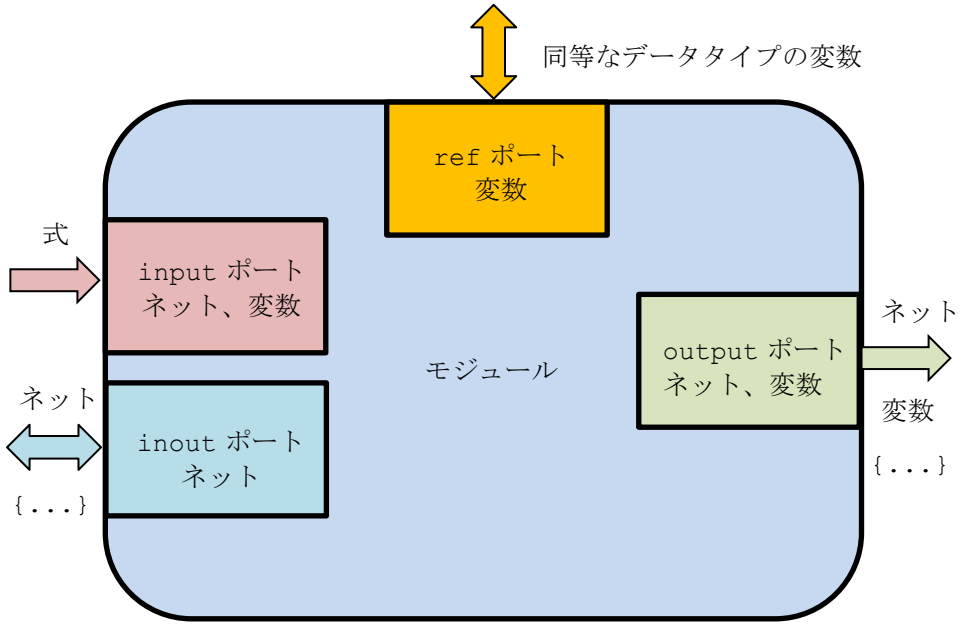


図 11-1　モジュールのポートと種類

12　ネットと変数　2021.10.05

変数は、一時的に値を確保するために使用され、ネットは常時接続されている状態を表現するために使用されます。

ネットは、要素同士を接続するために使用されるオブジェクトで、`trireg` を除き、値を保持する機能はありません。そのため、ネットの値はドライバーの値により決定されます。したがって、ネットには、常に、ドライバーが接続されていなければなりません。もし、ドライバーが接続されていない場合には、ネットの値として`'z` が仮定されます。ただし、`trireg` の場合には以前のネットの値が仮定されます。

参考 12-1

一般的に言って、ネットは複数のドライバーを持ち得ますが、変数はシングルドライバーに限られます。ただし、汎用的な `always` プロシージャは一つのグループとして考えられるので、変数が複数の汎用 `always` プロシージャでドライブされても一つのドライバーしか持たないと判定されます。なお、`always_comb` や `always_latch` は汎用 `always` プロシージャから除外されます。詳しい機能解説は、文献[1,9]を参照下さい。
□

13　レベルセンシティブとエッジセンシティブ　2021.10.05

ハードウェア記述言語に慣れない初心者には、レベルセンシティブとエッジセンシティブの違いはそれ程明確ではないと思います。下記の問題を解けば、差異を理解する事ができると思います。

問題 13-1

① 下記の記述を読んで、イベント待ちが解除されるか否かの結果を埋めて下さい。分からない場合には、シミュレータを使用して結論を出しても構いません。

記述	ベント待ちの解除結果
module test; event ev; initial begin　　レベルセンシティブ ->ev; **wait(ev.triggered);** $display("@%0t: main completed",$time); end endmodule	どちらかを選んでください。 □イベント待ちは解除する。 □イベント待ちは解除しない。
module test; event ev; initial begin ->ev;　　エッジセンシティブ **@ev;** $display("@%0t: main completed",$time); end endmodule	どちらかを選んでください。 □イベント待ちは解除する。 □イベント待ちは解除しない。

② 上記の問の解を踏まえて、レベルセンシティブとエッジセンシティブの差異を簡潔に説明して下さい。

■

14　プロセスの終了を待つ　2021.10.05

SystemVerilog で他のプロセスの終了を待つには、幾つかの方法があります。例えば、status()メソッドと await()メソッドの二つの手段があります。

表 14-1　status()と await()の比較

比較項目	status()メソッド	await()メソッド
記述法	process p; ... wait(**p.status** == process::FINISHED);	process p; ... **p.await**;
実行効率	プロセス p の状態が変化する度に、wait 文の再評価をするためのオーバヘッドがある。	プロセス p の処理が完了した時点で待ち状態が解除されるので、オーバヘッドは少ないと期待できる。

備考	status()メソッドは、wait 文と使用せずに、通常の実行文と使用する方が適切。
	
	task stop();
	　foreach(job[i]) begin
	　　case (**job[i].status()**)
	　　process::RUNNING,
	　　process::WAITING,
	　　process::SUSPENDED: job[i].kill();
	　　endcase
	　end
	endtask	

15　UVM に関するルール　2021.10.07

UVM を使用する際には、以下に示すルールを守らなければなりません。

表 15-1　UVM で検証環境を構築する基本的なルールと手順

検証環境構成種別	ルール
検証コンポーネント	• 検証用のコンポーネントを開発するためには、必ず、メソドロジークラス、または、そのサブクラスから定義しなければなりません。 • 代表的なメソドロジークラスとしては、uvm_driver, uvm_sequencer, uvm_monitor, uvm_agent, uvm_env, uvm_test, uvm_scoreboard 等があります。 • DUT との接続には virtual インターフェースが使用され、通常、ドライバー（uvm_driver のサブクラス）とコレクター（uvm_component のサブクラス）が DUT と接続されます。 • ドライバーは、DUT をドライブする役目を持ち、コレクターは DUT からのレスポンスをサンプリングする役目を持ちます。収集したレスポンスはトランザクションに変換された後、他の検証コンポーネントに送信されて、DUT の検証に使用されます。
トランザクション	• uvm_sequence_item、または、そのサブクラスからトランザクションを定義しなければなりません。 • DUT をドライブするために必要な情報を定義します。
検証手順（シナリオ）	• 検証手順とは、トランザクションを生成する手順でシーケンスにより表現されます。シーケンスを階層的に構築する事ができるので複雑なシナリオを表現できます。 • uvm_sequence、またはそのサブクラスからシーケンスを定義しなければなりません。 • シーケンスは、トランザクションを生成するための手順ですが、トランザクションを作るタイミングは、通常、ドライバーが決定します。 • シーケンサー（uvm_sequencer のサブクラス）は、ドライバーとシーケンスの仲介役を果たします。つまり、ドライバーはトランザクションを取得するためにシーケンサーと通信し、シーケンサーはシーケンスにトランザクションを作って貰います。 • 通常、テスト（uvm_test のサブクラス）がシーケンサーとシーケンスを結びつけます（default_sequence）。

参考 15-1

UVM には DUT からのレスポンスをサンプリングする機能を持つコレクターのメソドロジークラスが定義されていないので、コレクターのベースクラスを定義する事を勧めます。モニター内にコレクター機能を実装すると柔軟性に欠けると考えられます。
□

16　UVM シミュレーション実行に関するメソッドの呼び出し　2021.10.08

UVM では、`raise_objection()` と `drop_objection()` を呼ばなければ、シミュレーションは実行しません。これらのメソッドの呼び出し管理をする要素としては、以下の候補が考えられます。

- テスト（uvm_test のサブクラス）
- ドライバー（uvm_driver のサブクラス）

ドライバーが、メソッドの呼び出し管理をする事になると、それぞれのテストケースに関する知識が必要になる可能性があり、ドライバーの再利用性、汎用性、柔軟性に制限が出てきます。したがって、ドライバーが担当するのは望ましくないと思えます。

テストは、シーケンサーとシーケンスを結びつける役目を持ちますが、テストケース（シーケンスの事）の中身まで管理するのは行き過ぎだと思えます。したがって、テストよりもシーケンスにメソッドの管理を任せる方が適切です。

シーケンスは階層的に構築されるのでテストケースは多くのシーケンスから構成されますが、それぞれのシーケンスでメソッドの呼び出し管理をするのは複雑になるだけです。望ましい方法は、ルートシーケンスが開始する時に `raise_objection()` を呼び、ルートシーケンスが終了する時に `drop_objection()` を呼ぶ方法だと思います。

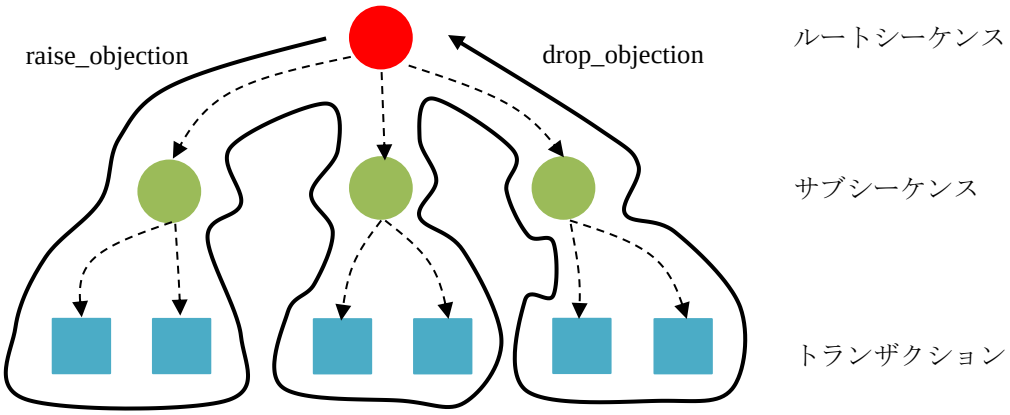


図 16-1　ルートシーケンスで実行制御する方法

17　C/C++と SystemVerilog の差異　2021.10.16

C/C++では、アレイや文字列はポインターで管理されますが、SystemVerilog ではそれらはポインターで管理されずにオブジェクトとして扱われます。例えば、文字列同士の比較が可能です。また、サブルーティンの引数にアレイを指定するとアレイのコピーが発生します。

SystemVerilog では、アレイや文字列はポインターではなく、オブジェクトとして扱われます。

```
string       state;
logic [1:0]  out;

always @(state)
   case (state)
   "HOLD":   out = 0;
   "RESET":  out = 1;
   "SET":    out = 2;
   "TOGGLE": out = 3;
   default   out = 'x;
   endcase
```

文字列を指定できる

```
function void check(input int a[],...,
                    output int x[]);
...
endfunction
```

ファンクションが開始する時、アレイ a へのコピーが発生する

ファンクションが終了する時に、アレイ x からのコピーが発生する

参考 17-1

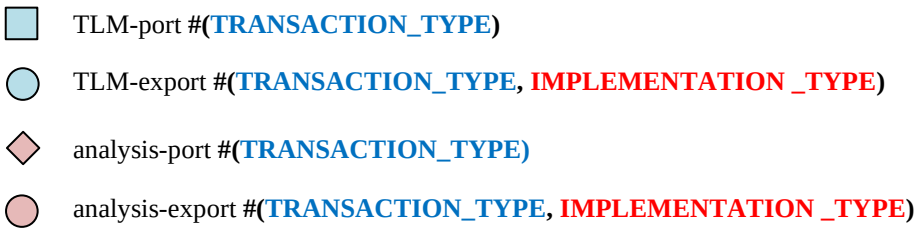
そもそも、string は byte 型の可変長アレイなので、通常のアレイと同じ扱いをされます。そして、文字列はポインターではないので、文字列の終了を示すナル（\0）は必要ありません。また、文字列にナルを含んでも無視されます。

□

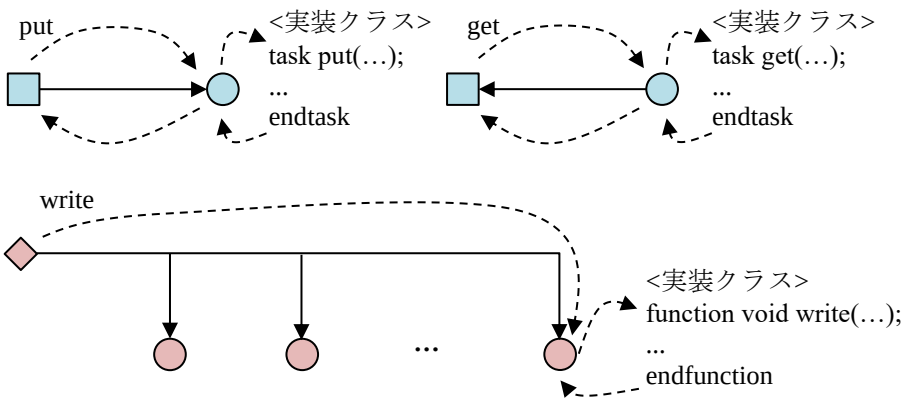
18　UVM の実装技術　2021.10.28

UVM に適用されている SystemVerilog の知識・技術は参考に値します。例えば、クラスタイプをパラメータに指定する事により、検証機能の汎用化が実現します。TLM は最たる例です。

UVM における通信は、通常、一対一で行われ、トランザクションが送信専用として使用される場合には、一対 N の通信が可能です。一対一の通信には、TLM-port と TLM-export が使用され、一対 N の通信には analysis-port と analysis-export が使用されます。一般に、port はトランザクション型の情報だけで十分ですが、export の場合はトランザクション処理を記述した実装コンポーネントの型も必要とします。しかし、RTL に比較すると接続は極めて簡単になります。



TLM 通信はポート同士（port と export）で行われますが、export の場合には実装コンポーネントの型情報が必要になります。実装コンポーネントクラスの型が分かれば、export はそのクラスに定義されている put や get 等のメソッドを呼ぶ事ができます。こうして、port と export 間の通信が可能になります。



19　ファンクショナルカバレッジとスケーラビリティ　2021.10.02

ファンクショナルカバレッジの仕様を定義する場合、SystemVerilog の知識を活用すればスケーラビリティを高める事ができます。

例えば、以下のようなカバレッジ定義を仮定します。

```
class item_t;
rand logic [3:0] a;

covergroup cg;
   coverpoint a { bins value[4] = { [0:15] }; }
endgroup
...
```

この例ではランダム変数 a のカバレッジを計算するためにカバーポイントとビンを定義しています。ランダム変数 a は 4 ビットなので、値域として[0:15]が指定されていますが、この指定法はスケーラブルではありません。このため、ランダム変数 a のビット数に変更が生じた場合には、以下のような問題が出てきます。

- 変数 a のビット数を変更するとビンの定義も変更しなければならない。
- ビット数が増加すると最大数を簡単に思いつかない。

寧ろ、以下のような指定をすればランダム変数 a のビット数には依存しなくなり、記述法はスケーラブルになります。

```
coverpoint a { bins value[4] = { [0:$] }; }
```

あるいは、以下のようにする事もできます。

```
class item_t;
parameter NBITS = 4,
          LEFT = 0,
          RIGHT = $;
rand logic [NBITS-1:0] a;

covergroup cg;
   coverpoint a { bins value[4] = { [LEFT:RIGHT] }; }
endgroup
...
```

20　算術式の型　2021.11.10

SystemVerilog では、式の型はオペランドの型で決定されるので、予期しない結果が得られる事があります。例えば、`2.0**-1` は `0.5` ですが、`2**-1` は `0` になります。

SystemVerilog LRM（256-257 ページ）には、以下の様に規約が定められています。

> The result of using logical or relational operators or the inside operator on real operands shall be a single-bit scalar value.
>
> For other operators, if any operand, except before the ? in the conditional operator, is real, the result is real. Otherwise, if any operand, except before the ? in the conditional operator, is shortreal, the result is shortreal.

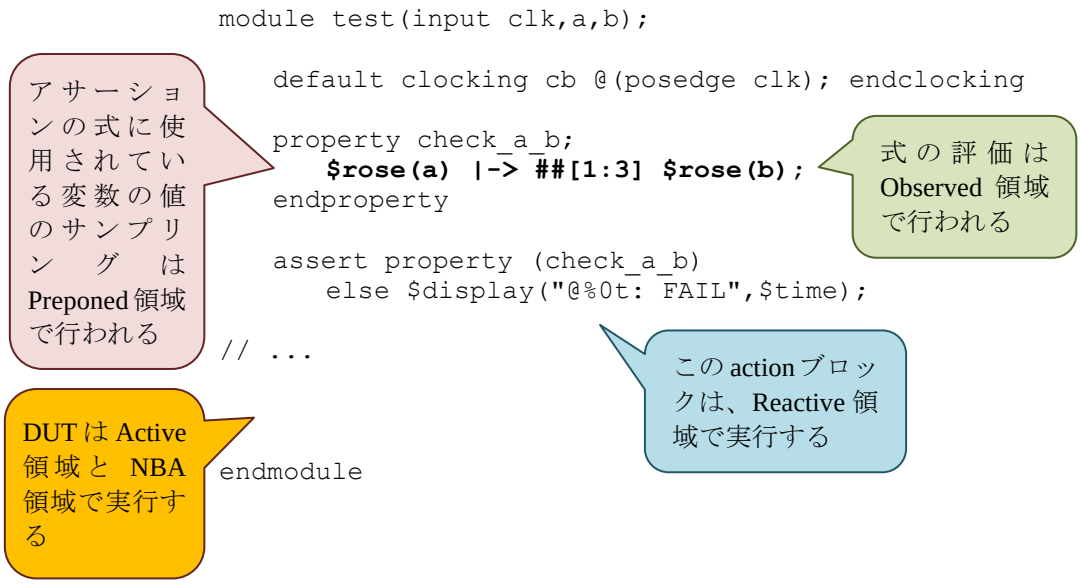
簡単に言えば、ビット演算や比較演算を除き、算術式のオペランドに実数型が使用されれば、式の型は実数型となります。それ以外の場合には、整数型（integral）になります。

この規約に従うと、**`2.0**-1`** は実数型なので `0.5` となりますが、**`2**-1`** は整数型なので切り捨てられて `0` となります。

また、**`9**0.5`** は `3.0` ですが、**`9.0**(1/2)`** は `9.0**0` を意味するので `1.0` となります。僅かな違いで結果が全く異なるので注意が必要です。

21　アサーション　2021.11.12

SystemVerilog のアサーションは仕様が複雑である事に加えて、実行状況も複雑である事がアサーションの適用を妨げているかも知れない。尤も、誰もがアサーションを理解する必要はないのですが。



22　コンポーネントインスタンス　2021.11.13

検証コンポーネント間で通信をする場合、相手側のコンポーネントのインスタンスを保有するのは望ましくない。寧ろ、ポートを介して通信するのが近年の検証法と言える。コンポーネントのインスタンスは、あくまで検証機能を階層化（即ち、細分化）する手段とするのが最近の流儀（例：UVM）。

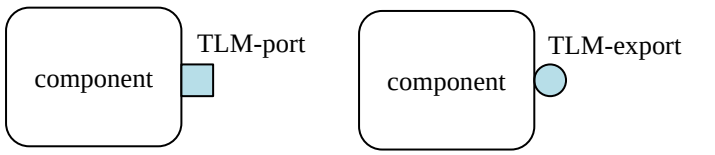
UVM は、virtual メソッドを活用して検証要素の柔軟性・汎用性を高めていると言えます。クラスインヘリタンスの技術と効果は OOP から容易に学べますが、UVM からはそれを超える技術と手法を学ぶ事ができます。

UVM の検証コンポーネントは、トランザクションを送受信するための通信対象を持ちますが、それが明示的に指定される事はありません。これにより、検証コンポーネントの再利用性が高まります。

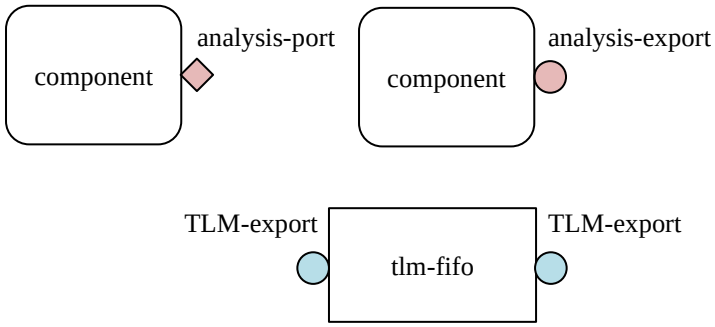
検証コンポーネントの通信対象を決定するのは、階層上の親コンポーネントです。したがって、UVM には以下に示す原則があります。

- 各検証コンポーネントは親コンポーネントを持つ
- 階層構造をトップダウンで構築する
- 階層構造を構築時に通信対象を決定し、別のフェーズで接続を完了する
- 接続を完了するフェーズはボトムアップで行われる

検証コンポーネントにはポートを定義して通信を可能にします。但し、通信対象のコンポーネントを指定しません。



検証コンポーネントが親コンポーネントに配置された後に別のフェーズで、ポート接続が完了されます。この手法により、検証環境をダイナミックに決定する事ができます。

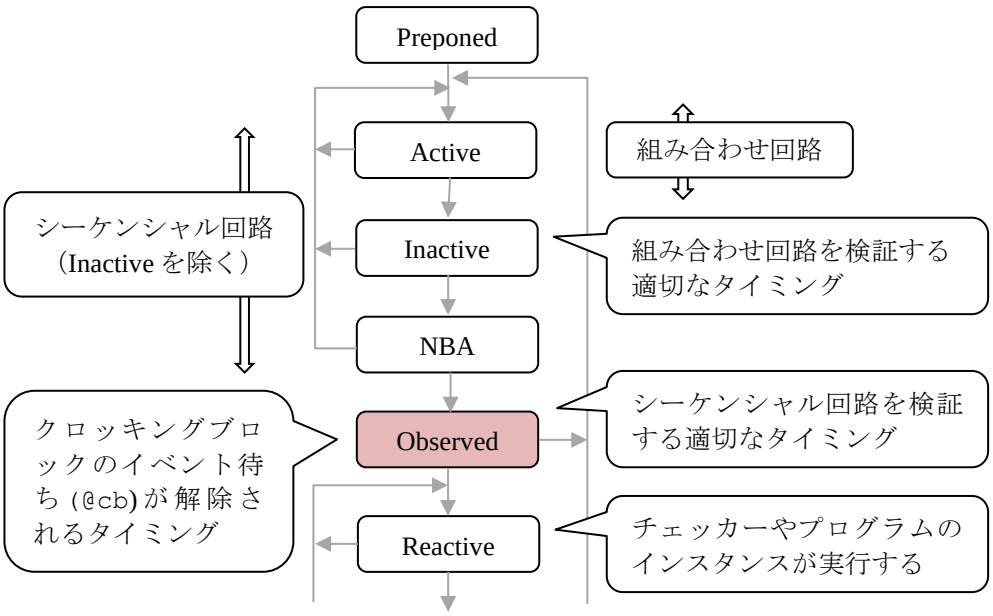


23　検証環境の構築とアプローチ　2021.11.18

SystemVerilog のファンクショナルカバレッジやアサーション等を含む複雑な検証環境では、それぞれの機能が実行するタイミングを正確に理解しておかなければなりません。例えば、トランザクションが実際に使用された後にのみカバレッジ計算に意味がある場合には、Observed 領域以降で有益です。

チェッカーやプログラムを利用するためには、スタティックにインスタンスを作る必要があります。このため、これらを使用した検証環境はダイナミック性に欠けるので望ましくありません。寧ろ、クロッキングブロックとクラスを使用したダイナミックな検証環境の方が近年の検証技術に適合し易いと言えます。

即ち、UVM とクロッキングブロックを使用した検証環境を構築し、Observed 領域で DUT からのレスポンスをサンプリングすれば、現代的な検証法を実現できると言えます。



24　代入文　2021.12.03

代入文において、右辺のビット長が左辺のビット長よりも大きい場合、右辺の余分な MSBs は無視されます。右辺の符号も失われるので、左右のビット長を合わせておくのが安全と言えます。

LRM 244 ページより抜粋

```
logic [0:5]         a;
logic signed [0:4]  b, c;

initial begin
   a = 8'sh8f;   // After the assignment, a = 6'h0f
   b = 8'sh8f;   // After the assignment, b = 5'h0f
   c = -113;     // After the assignment, c = 15
                 // 1000_1111 = (-'h71 = -113) truncates
                 // to ('h0F = 15)
end
```

25　初期化　2021.12.5

固定サイズのアレイに同じ値を設定する場合には、default を使用したアレイリテラルが便利です。例えば、以下のようにすると、アレイ a の全ての要素に同じ値を設定する事ができます。しかも、アレイのサイズには依存しないので、スケーラブルな記述になります。

```
logic [3:0]  a[5];
string       color[7];

a = '{default:2};              // 全てのアレイ要素に 2 を設定する。
color = '{default:"red"};      // 全てのアレイ要素に"red"を設定する。
...
```

以下のようにも表現する事ができますが、アレイのサイズに変更が生じると正しく動作しなくなります。

```
a = '{5{2}};
```

あるいは、以下のように初期化をする事ができますが、アレイリテラルに比べると実行効率は良くないかも知れません。

```
foreach(a[i])
   a[i] = 2;
```

参考 25-1

以下に使用されている default の意味は若干異なります。前者の default は、アレイを初期化する際に使用されます。つまり、「インデックスが 1 以外の場合は」という意味です。後者は、指定したキーを持つアレイ要素が存在しない時に仮定する初期値を意味します。つまり、所謂、標準値を意味します。

```
logic [3:0]  a[5] = '{ 1:5, default:8 };
string       words[int] = '{ default:"hello" };
```

□

要点を以下にまとめておきます。

unscalable	scalable
logic [15:0] value; value = -1;	logic [15:0] value; value = '1; value = '{ default:1 };
byte a1[5]; a1 = '{ 5, 10, 10, 10, 10 }; a1 = '{ 5, 4{10} };	byte a1[5]; a1 = '{ 0:5, default:10 };
string colors[5]; colors = '{ "red", "red", "red", "red", "red" }; colors = '{ 5{"red"} };	string colors[5]; colors = '{ default:"red" };

26　concatenation オペレータ　2021.12.06

SystemVerilog の concatenation オペレータ{}を使用して生成されたビット列を packed アレイのように扱う事ができます。したがって、ビットスライスも可能です。例えば、{a,b}[1:0]や{a+b}[1:0]のような記述をする事ができます。ただし、このような記述法を式の左辺に指定する事はできません。

以下の例を参考にして下さい。

```
module test;
byte      a, b ;
bit [1:0] c ;

initial begin
   a = 2;                  // 0000_0010
   b = 4;                  // 0000_0100
   c = {a + b}[1:0];       // 2 lsb's of sum of a and b
   $display("c = 'b%b",c); // 0000_0110 -> 10
end

endmodule
```

実行結果は以下のようになります。

```
c = 'b10
```

27　SystemVerilog の計算精度決定法　2021.12.07

SystemVerilog の演算では、演算結果の精度を保証するように計算が行われます。精度の決定にはオペランドのビット長だけでなく、左辺のビット長も考慮されます。

例えば、SystemVerilog の加算（＋オペレータ）では、両オペランドに加えて左辺も考慮して、それらの中で最も大きいビット長を基準にして演算精度を決定します。

SystemVerilog LRM 283 ページより以下の例を抜粋して解説します。

```
logic [15:0] a, b;  // 16-bit variables
logic [15:0] sumA;  // 16-bit variable
logic [16:0] sumB;  // 17-bit variable

sumA = a + b;       // expression evaluates using 16 bits
sumB = a + b;       // expression evaluates using 17 bits
```

sumA を求めるための加算(a + b)は 16 ビットで行われるので、加算でオーバーフローが発生すれば、計算結果は正しく求められません。一方、sumB を求めるための加算(a + b)は 17 ビットで行われるので、加算でのオーバーフローも考慮されているため、sumB には正しい結果が求まります。このように、SystemVerilog では記述に従って計算精度を保証するような仕組み採用しています。

参考 27-1

更に理解を深めるためには、e.sum に関する以下の挙動の差異を解析する事を勧めます。

```
logic [3:0]  c;
bit          e[5] = '{ 1, 1, 1, 0, 0 };

initial begin
   $display("e.sum = %0d",e.sum);
   $display("e.sum = %0d",e.sum+0);
   c = e.sum;
   $display("c = %0d",c);
end
```

□

28　SystemVerilog の計算精度決定法（続き）　2021.12.08

以下の解説は、LRM 284 ページからの抜粋です。仮に、三つの変数 a、b、answer が次のように宣言されていたとします。

```
logic [15:0] a, b, answer;    // 16-bit variables
```

以下のような計算をしたとします。一見、この記述は正しく見えますが、正しい計算結果を導けない場合があります。

```
answer = (a + b) >> 1;        // will not work properly
```

全ての変数は 16 ビットなので、(a + b) は 16 ビットの精度で計算されます。したがって、もし加算でオーバーフローが発生した場合には、キャリービットの情報が失われる事になり、正しい計算結果を得る事ができません。

正しい計算結果を求めるには、(a + b) の計算を 17 ビット以上で行わなければなりません。以下のようにすれば、この問題を解消する事ができます。

```
answer = (a + b + 0) >> 1;    // will work correctly
```

参考 28-1

シフトオペレータ（>>）では、左オペランドが精度を決定します。右オペランドの定数 1 は計算精度に影響を持ちません。
□

参考 28-2

C/C++では、(a + b + 0) と記述しても、(a + b) と最適化されます。しかし、SystemVerilog では、このような最適化をする事は一切できません。
□

29 SystemVerilog の計算精度決定法（まとめ） 2021.12.09

主要なオペレータに対してビット長の計算法を下表に紹介します。表において、L(i)はオペランド i のビット長を示します。

主要なオペレータと演算精度の決定法（LRM 283 ページの表を編集）

演算式	op	演算精度
i op j	+ - * / % & | ^ ^~ ~^	max(L(i),L(j))
	=== !== == != > >= < <=	1 bit
	&& ||	
	>> << ** >>> <<<	L(i)
i ? j : k	conditional	max(L(j),L(k))
{i,...,j}	concatenation	L(i)+...+L(j)
{i{j,...,k}}	replication	i ×(L(j)+...+L(k))

?:オペレータには注意が必要です。下記の(a&b)の精度は 5 ビットになります。

```
logic [3:0]  a, b, c;
logic [4:0]  d;

initial begin
   a = 9;
   b = 8;
   c = 1;
   $display("answer = %b", c ? (a&b) : d);
end
```

d が 5 ビットなので、(a&b)は 5 ビットに拡張されるため、結果は **5'b0_10000** になります

更に、以下の例は驚きを与えます。

```
logic [3:0]  a;
logic [5:0]  b;
logic [15:0] c, d;

initial begin
   a = 4'hF;
   b = 6'hA;
   $display("a*b=%h", a*b);
   c = {a**b};
   d = a**b;
end
```

a*b は 6 ビットなので、'h96 ではなく **6'h16** となります

{a**b}は 4 ビットなので **c** は **16'h0001** となり、**d** は **16'hac61** となります

以上の例を踏まえて、以下に経験則をまとめておきます。

- {...}の精度はコンパイル時に決定されなければならないため、{...}の中に演算式を記述しても精度は拡張されません。したがって、この中には演算式を記述しないようにすると問題を未然に防ぐことができます。
- expr ? a : b の精度は、max($bits(a),$bits(b))で決定されます。したがって、$bits(a)と$bits(b)が異なる場合には注意が必要です。
- 演算結果がオペランドのビット長を超える可能性があるオペレータ（+、*、**、等）の場合には、精度が十分に保証されるように記述する必要があります。

30　{} オペレータと繰り返し　2021.12.16

SystemVerilog の {} オペレータの演算優先順位は最も低く設定されています。したがって、反直感的ですが、`{5-1{2'b10}}`は`{4{2'b10}}`に等しくなります。つまり、繰り返しの値が式で表現されていると、式の評価が先に行われます。

31　多才な {} オペレータ　2021.12.19

SystemVerilog のオペレータの中で、{} は最も多才なオペレータの一つと言えます。その多才さを復習してみます。

- 基本的な機能として、{} は packed アレイを作ります。

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

- {} の結果は packed アレイなので、パートセレクトが可能です。

```
{a,b}[7:4]
```

- {} の結果は unsigned なので、{} は unsigned に変換する機能を持ちます。

```
{-1}-1
```

は、4294967294 となります。ただし、ここでは-1 が 32 ビットで表現されていると仮定しています。このように {} オペレータを unsigned に変換する機能として使用できます。パートセレクトも併用できるので、非常に便利です。

- ビット長を拡張せずに演算を実行します。

例えば、{a+b}のビット長は max($bits(a),$bits(b))なので、オーバーフローは無視されます。{a*b}も同様です。また、{a**b}のビット長は$bits(a)と等しいため、溢れた MSBs は切り捨てられます。

- 文字列を結合する機能を持ちます。

```
string    hello = "hello",
          tag = {5{"*"}};
string    s;
s = { tag, " ", hello, " ", "world ", tag };
```

- キューを初期化するためのキューリテラルとして {} が使われます。

```
byte   q[$];
q = { 10, 20, 30 };
```

32　パートセレクト　2021.12.28

SystemVerilog のパートセレクトは、常に符号なしです。たとえパートセレクトが符号付きの packed アレイ全体を示していても符号は無視されます。

例えば、以下のように変数 b は符号付きで宣言しても、b[7:0]には符号が付いていません。

```
module test;
logic [15:0]        a;
logic signed [7:0]  b;

initial begin
   b = -1;
   a = b;           // a = 'hffff
   a = b[7:0];      // a = 'h00ff
end

endmodule
```

33　SystemVerilog と最適化　2021.12.30

C/C++では、式(a+b+0)を(a+b)に最適化できますが、SystemVerilog ではその様な最適化をできません。この場合には、強制的に 32 ビット以上で加算をするために 0 を指定しています。即ち、意図的に 0 を指定しているので、SystemVerilog ソースコードの最適化には注意が必要です。

最適化すべきかすべきではないかの問題

SystemVerilog では全く最適化する機会がないかと云えば、そうでもありません。寧ろ、最適化が必要な場合もあります。先ず、既に紹介したように、次の記述では最適化できません。1 ビット右にシフトするので、a+b の加算は 17 ビット以上で行われなければなりません。そのためには、0 が必要です。

```
logic [15:0] a, b, answer;
...
answer = (a + b + 0) >> 1;
```

(a + b)>>1 に最適化できない

以下の場合では、b と b[7:0]の内容は同じですが、b[7:0]を b に最適化する事はできません。b は符号付きですが b[7:0]は符号なしなので、論理的に意味が異なるからです。

```
logic [15:0]         a;
logic signed [7:0]   b = -1;

a = b[7:0];   // a = 'h00ff
a = b;        // a = 'hffff
```

b[7:0]を b に最適化できない

最適化できない例をもう一つ紹介します。アレイ e の要素は 1 ビットなので、e.sum の計算結果は 1 ビットになります。正しく求めるためには、e.sum+0 を最適化できません。

```
bit    e[5] = '{ 1, 1, 1, 0, 0 };
...
$display("e.sum = %0d",e.sum);    // 1 がプリントされる
$display("e.sum = %0d",e.sum+0);  // 3 がプリントされる
```

一方、以下の例で、もし tmp が他の何処でも使用されていなければ、tmp を最適化できます。

```
logic  tmp;

always @(posedge clk) begin
    tmp = a&b;
    q <= tmp;
end
```

結果は、以下のようになります。最適化が望ましい一例です。

```
always @(posedge clk)
    q <= a&b;
```

34　logic データタイプ　2022.01.15

SystemVerilog の logic は、最も誤解し易い SystemVerilog 機能の一つだと思います。logic はデータタイプであり変数を宣言するためのマジックではありません。ネットでも変数でも logic 型になり得ます。

以下の表に示すように `logic` はデータタイプです。しかし、モジュールポートの宣言に関しては、SystemVerilog には特別なルールがあるため、ネットと変数の識別には確実な知識が求められます。正しくデザインを記述するためには、このルールを完全に理解しなければなりません。以下の例にあるように、ネットに対しても `logic` を指定できます。

宣言	ネット	変数
module dut(input a,b,output co,sum);	a, b, co, sum	
module dut(input a,b,output **logic** co,sum);	a, b	co, sum
logic a, b; wire **logic** co, sum; var x; var **logic** y;	co, sum	a, b, x, y

ネットタイプ（`wire`,`wand`,`wor`,`uwire`,`trireg` 等）を指定すると、必ず、ネットになりますが、ネットタイプを省略すると宣言状況によりネット、または変数になります。例えば、以下の文献により理解を深める事ができます。

SystemVerilog LRM（**IEEE Std 1800-2017**）、85―101 ページ、703―705 ページ
SystemVerilog 入門（共立出版）、24―31 ページ、272―273 ページ

35　{} オペレータと string　2022.01.18

SystemVerilog の {} オペレータのオペランドはコンパイル時にビット数が確定していなければなりません。一方、{} のオペランドとして string 型も許されますが、string 型オペランドの長さは可変です。これは {} オペレータの前提条件に矛盾するように思えますが、そうでもありません。

SystemVerilog のルール

string 型のオペランドが {} オペレータに使用されると、前提条件に矛盾するため、SystemVerilog では以下のような規約を適用します。

{} の少なくとも一つのオペランドが string 型であれば、{} は string の結合機能と判断され、string 型ではないオペランドは string 型に変換されます。つまり、{} オペレータの結果は string 型になります。

実は、次のように、このルールは非常に重要な拡張機能を意味します。

{} のオペランドには繰り返しが許されますが、一般に、繰り返しを示す数は定数式に限られます。しかし、string 型のオペランドは可変長なので、string 型の {} オペレータでは、繰り返し数を定数式に限定する理由はありません。すなわち、string 型 {} オペレータ内の繰り返しを示す式として**定数式以外**を指定する事ができます。

{} オペレータと繰り返し指定の例

integral 型 {} オペレータでは、繰り返しは定数式でなければなりません。

```
byte       a;
logic      b;
...
a = {8{b}};
```

繰り返しには定数式しか許されない

string 型 {} オペレータでは、繰り返しは non-constant でも構いません。

```
int    n;
string s
...
n = 3;
s = {n{ "boo " }};
$display( "%s", s ); // displays 'boo boo boo '
```

繰り返しには定数以外も許される

string 型 {} を代入文の左辺に指定する事はできません。

36　foreach　2022.01.21

SystemVerilog でループ処理を書く時、foreach と for ループの選択肢がある場合、foreach の使用は自明な間違いを未然に防ぐ効果があります。

簡単な例を使用して foreach が自明な間違いを未然に防ぐ効果がある事を紹介します。先ず、以下のようなファンクションを仮定します。このファンクションは、文字列アレイにおける文字列の最大長を求めています。

```
function int max_len(input string a[]);
    max_len = 0;
    foreach(a[i])
        if( max_len < a[i].len )
            max_len = a[i].len;
endfunction
```

この処理を for ループで実現すると、例えば、以下のようになります。

```
function int max_len(input string a[]);
    max_len = 0;
    for( int i = 0; i < a.size; i++ )
        if( max_len < a[i].len )
            max_len = a[i].len;
endfunction
```

しかし、max_len の初期化が for ループ外にあるので、うっかり次のように変更してしまう可能性は皆無ではありません。

```
function int max_len(input string a[]);
    for( int i = 0, max_len = 0; i < a.size; i++ )
        if( max_len < a[i].len )
            max_len = a[i].len;
endfunction
```

このように記述すると、max_len は for ループ内に確保されるローカル変数となり、ファンクションの戻り値を示す変数 max_len とは無関係です。この記述は明らかに間違いですが、コンパイルエラーが出ないのでその間違いに気づかずに悩み続ける事は確実です。もし、最初から foreach で記述していれば、このような問題は決して起こらないと言えます。

37　同じサイズを持つ packed アレイの入れ替え　2022.01.31

一般のプログラミング言語の習慣が、良くも悪くも SystemVerilog のプログラミングに無意識に影響を及ぼしています。例えば、二つの数値をスワップする場合、中間変数を使用する習慣がついています。然し、SystemVerilog の利点を生かすと他の選択肢があります。

同じサイズを持つ二つの packed アレイの内容を入れ替える処理は複雑ではありませんが、記述する順序に神経を使う事は確かです。以下の記述を比較して下さい。特別な理由がないのであれば、中間変数の使用を避けた方が楽です。

{} オペレータを使用したスワップ	中間変数を使用したスワップ

logic [7:0] x, y;
...
{x,y} = {y,x};
	logic [7:0] x, y, t;
	...
	t = x;
	x = y;
	y = t;

シーケンシャル回路の記述では、以下のように x と y を入れ替えますが、上記の場合にこの方法を適用すると、x と y の入れ替えが完了するタイミングの調整が複雑になるだけです。

```
x <= y;
y <= x;
```

この他にも方法があるかと問われれば、Yes です。多少の技巧を伴いますが、以下の方法でも x と y を入れ替えできます。しかも、join の終了後には入れ替えが完了しています。

```
fork
   x = #0 y;
   y = #0 x;
join
```

この方法は非常に汎用的で、任意のイベント発生時に入れ替えを行える利点があります。以下の例を参考にして下さい。

```
`define   sync_swap(n)  repeat(n) @(posedge clk)
...
fork
   x = `sync_swap(3) y;
   y = `sync_swap(3) x;
join
```

以上のように、幾つかの方法がありますが、特別な制約がなければ {} オペレータが最も簡単であると思えます。

38　$による最大・最小値の表現　2022.02.09

SystemVerilog では、$を活用すると良いです。区間の最大・最小値はビット数に依存するので、記述を汎用的にするためには$は有効です。

SystemVerilog の inside オペレータは、以下のようなシンタックスを持ちます。

```
inside_expression ::= expression inside { open_range_list }
```

inside オペレータの右辺には区間を指定できます。$を区間の左に指定すると、expression の最小値に、$を区間の右に指定すると expression の最大値になります。

以下の例では、指定した値が区間[-8:2]、または区間[5:7]に属するかをテストします。下記の記述例では-8 と 7 の代わりに$を使用しています。

```
typedef logic signed [3:0] small_t;

module test;

initial begin
    check(5);
    check(-5);
    check(3);
end

function void check(small_t v);
    if( v inside { [$:2], [5:$] } )
        $display("%0d is ok",v);
    else
        $display("%0d is illegal",v);
endfunction

endmodule
```

39　モジュールのポート宣言とネットと変数　2022.02.11

SystemVerilog では、オブジェクトはネットか変数の何れかです。通常の宣言では、ネットか変数の識別は容易ですが、モジュールヘッダのポート宣言となると区別し難くなります。初心者にとっては難関の一つで、SystemVerilog を好きになるか嫌になるかの瀬戸際です。

先ず、ポートの種類を定義しておきます。ポートの種類とは、ネット型のキーワード、またはキーワード var を意味します。ポートの種類が指定されていれば、ネットか変数であるかは自明です。例えば、以下の宣言でポート x は変数、ポート y はネットになります。

```
module m(input var integer x,wire y);
```

問題はポートの種類が省略された場合ですが、その場合には以下のルールが適用されます。

ポートの種類が省略された場合のルール

ポートの方向	仮定されるポートの種類
input	標準ネット型のネット。
inout	
output	データタイプが省略されるか、または implicit_data_type が指定されれば、標準ネット型のネットに設定されます。もし、データタイプが指定されれば、変数になります。
ref	常に、変数です。

ここで、implicit_data_type は以下のように定義されています。

```
implicit_data_type ::= [ signing ] { packed_dimension }
signing ::= signed | unsigned
```

以下の宣言例を見て下さい。これを理解すると SystemVerilog の学習は楽しくなります。

モジュールヘッダ	ネット・変数
module m1(logic x);	x：inout のネット ポートの方向が指定されていないので、x は inout になります。
module m2(output integer x);	x：output の変数
module m3(ref [5:0] x);	x：ref タイプの変数
module m4(integer x,signed [5:0] y);	x,y：inout のネット
module m5(output signed [5:0] x,integer y,z);	x：output のネット y,z：output の変数 z は y の属性を継承します。

参考文献
SystemVerilog LRM（IEEE Std 1800-2017）、85―101 ページ、703―705 ページ
SystemVerilog 入門（共立出版）、24―31 ページ、272―273 ページ

40　a[n:m]　2022.02.19

SystemVerilog では、a[n:m]と書くと 3 通りの解釈があります。

SystemVerilog では、packed アレイ、unpacked アレイ、キュー等があるため、それぞれの構造に対して記法 a[n:m]が許されています。下表はそのまとめです。

用法	機能	記述例
パートセレクト	• 1 次元 packed アレイの連続したビットを参照する仕組みです。 • a[n:m]において、n と m は定数式に限ります。	logic [63:0]　data; logic [7:0]　b; b = **data[23:16];**
アレイスライス	• 連続したアレイ要素を参照する機能です。 • a[n:m]において、n と m は定数式に限ります。	bit signed [31:0] busA [7:0]; int　busB [1:0]; busB = **busA[7:6];**
キュースライス	• 連続したインデックスを持つキュー要素からサブキューを作る機能です。 • q[n:m]において、n と m に対して一般の計算式を適用できます。	byte　q[$] = { 1, 2, 3 }, 　　r[$]; r = **q[1:2];**

41　曖昧な if 文シンタックス　2022.02.24

コンパイラーの分野で、if 文のシンタックス上の曖昧性を回避する議論が良くされますが、SystemVerilog の if 文についても同様の注意が当てはまります。

先ず、SystemVerilog の if 文のシンタックスは以下のように定義されています。

```
[ unique_priority ] if ( cond_predicate ) statement_or_null
{ else if ( cond_predicate ) statement_or_null }
[ else statement_or_null ]
```

実は、このシンタックスは曖昧です。例えば、以下の例を考えてみて下さい。ここで、e1、e2 は条件式で、stm1、stm2 は文とします。

```
if( e1 ) if( e2 ) stm1 else stm2
```

この記述は、else をどちらの if と結合するかで意味が変わってきます。以下の(a)と(b)のように二通りの解釈が可能です（インデンテーションで解釈の相違を表現しています）。

```
if( e1 )
    if( e2 )
        stm1
    else
        stm2
```

(a)

```
if( e1 )
    if( e2 )
        stm1
else
    stm2
```

(b)

```
if( e1 )
    begin
        if( e2 )
            stm1
    end
else
    stm2
```

(c)

SystemVerilog では(a)の解釈を採用します。つまり、else は else を持たない最も近い if に結合されます。(b)のように解釈させるためには、上記の(c)のように begin-end を使用しなければなりません。

曖昧性を除去した文法に書き直す方法は、文献[1,2]を参照下さい。

参考文献

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, Compilers : Principles, Techniques, and Tools, Addison-Wesley, 1998、174-175 ページ　← Red Dragon Book

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, Compilers : Principles, Techniques, and Tools, Addison-Wesley, 2007、210-211 ページ　← Purple Dragon Book

42　非推奨機能　2022.04.18

SystemVerilog には、使用しない方が良い機能もあります。

SystemVerilog 豆知識

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

43　randc ランダム変数　2022.04.24

SystemVerilog の randc ランダム変数にはランダム順列用の領域が確保されます。ランダム変数のビット数が増加すると、必要なメモリーも増加します。

SystemVerilog では、randc のランダム変数には以下の手順が適用されます。①は一度だけで、②と③は適宜繰り返されます。

① ランダム順列用の領域を確保する。
② ランダム順列を準備する。
③ クラスの randomize()メソッドが呼ばれると、ランダム順列から順に値を取りだして戻す。もしランダム順列から全ての値が取り出されてしまっていれば、新たなランダム順列を作ってから、順に値を取り出して同様の処理を繰り返す。

例えば、以下のように randc ランダム変数を宣言すると、長さが 256 の順列を格納するための領域が必要になります。ツールは何らかの方法で 256 個の相異なる数を記憶しなければなりません。加えて、乱数を順に取り出すための管理情報も必要になります。

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

ランダム順列用の領域を確保する負荷に加え、ランダム順列を準備する処理時間も必要になります。以上から、randc ランダム変数は rand ランダム変数よりもオーバヘッドが大きい可能性があります。

randc ランダム変数のビット数 n が 8 を超えると、実行効率は低下すると予想されるため、EDA ツールは n に関する制限を設けていると考えておいた方が安全です。因みに、SystemVerilog LRM では、n は少なくとも 8 まではサポートして欲しいと明記しています。

参考
以下の enum タイプの型は 32 ビットの int 型ですが、値としては、0、1、2 の三通りしか取り得ないので、生成されるランダム順列の長さは、常に、3 です。

```
typedef enum { RED, GREEN, BLUE }    rgb_e;
randc rgb_e      rgb;
```

従って、enum タイプの変数に randc を指定するのは、合理的であると考えられます。
□

44　reg その他　2022.05.04

SystemVerilog では reg を使用する必要はありません。また、logic と reg が全く同等という訳ではありません。

SystemVerilog 豆知識

機能	仕様と注意点
`logic`	• logic はデータタイプであり、それ以上の機能を持ちません。 • logic は変数を宣言する機能を持つわけではないので、注意が必要です。ネットでも変数でも logic 型を持ち得ます。 • データタイプが指定されずにネットや変数が宣言されると標準データタイプとして logic が仮定されるだけです。

`reg`	• SystemVerilog では、reg を使用する必要はありません。
	• logic と reg は全く同等とは言えません。ネットタイプと logic を一緒に指定できますが、以下のようにネットタイプと reg が一緒に使用されるとエラーになります。
	`tri reg r; // error`
	`wire reg [15:0] v; // error`

`signed` `unsigned`	• SystemVerilog では、unsigned と signed の比較は unsigned で行われます。例えば、`{-1}==-1` は unsigned 同士の比較になるので真です。
ビットセレクト パートセレクト	• ビットセレクトとパートセレクトは、常に、符号なしです。 • 例えば、以下のように宣言すると、`a[1]` も `a[15:8]` も符号なしです。 `logic signed [15:0] a;`
サイズを指定しない数値リテラル	• サイズを指定しない数値リテラル（`100`、`'habcd`、`'b10`）の占めるビット数は、少なくとも 32 ビットです。 • 実際のビット数は、EDA ツールに依存します。したがって、一般的には、`$bits(100) >= 32` となります。
`'0`、`'1`、`'x`、`'z`	• これらのリテラルは可変長ですが、単独では 1 ビットです。例えば、`$bits('0)==1` です。しかも、符号なしです。拡張されるビット数には制限がありません。 • Verilog-95 では、`'bx` や`'bz` は 32 ビットまでしか拡張されません。

45　論理合成可能な記述と自由度　2022.05.12

SystemVerilog/Verilog で記述する際、論理合成による記述ルールの習慣が弊害をもたらす事もあります。例えば、不完全な if 文を書くとラッチが生成されるため、一般的な検証問題でも不完全な if 文の記述を極力避ける習慣がつくのも不思議ではありません。

例えば、変数の内容を左または右にシフトし、端から溢れたビットを反対側に回り込ませる操作を考えてみます。

左または右にシフトする場合、シフトするビット数が定数であれば、単なる rewire になるのでコンパクトなネットリストになりますが、シフトするビット数が変数で与えられると、一般的なシフターが合成されるため、複雑な合成結果となります。ましてや、ビットシフトにより溢れたビットを反対側に回り込ませる仕様であれば、記述も定数方式にならざるを得ません。然し、これは論理合成可能性を考慮するために引き起こされる問題であり、この習慣が過度についてしまうと、一般的な検証問題への処理も必要以上に複雑になりかねません。検証では、自由な発想で問題を解く事ができます。下記の記述は、比較の一例です。

論理合成を考慮した安全な記述	自由な発想による記述
module shifter(input [7:0] data_in,[2:0] shift, 　output logic [7:0] data_out); always @(data_in,shift) 　case (shift) 　0: data_out = data_in; 　1: data_out = {data_in[6:0],data_in[7]}; 　2: data_out = {data_in[5:0],data_in[7:6]}; 　3: data_out = {data_in[4:0],data_in[7:5]}; 　4: data_out = {data_in[3:0],data_in[7:4]}; 　5: data_out = {data_in[2:0],data_in[7:3]}; 　6: data_out = {data_in[1:0],data_in[7:2]}; 　7: data_out = {data_in[0],data_in[7:1]}; 　endcase endmodule	module shifter(input [7:0] data_in,[2:0] shift, 　output logic [7:0] data_out); logic [7:0]　spill; always @(data_in,shift) begin 　{spill,data_out} = {8'b0,data_in} << shift; 　data_out |= spill; end endmodule

左側の記述は、左にビットをシフトする仕様ですが、右にシフトするためには、注意深く書き換えなければなりません。一方、右側の記述では、右シフトへの変更は容易です。

46　ツールの完成度と演算精度　2022.05.15

SystemVerilog の演算精度に関するルールは既に紹介しましたが、記述者は確実に機能するように準備する方が安全です。

例えば、以前の記述を例にとり話を進めます。

先ず、以下の式における演算精度は、SystemVerilog　のルールにより data_in の精度で決定されるので 8 ビットになります。

```
data_in << shift
```

この演算精度は、変数 shift に依存しない事に注意して下さい。したがって、shift の値が 0 でない場合、左にシフトして溢れたビットは失われてしまいます。これでは仕様を満たさないため、工夫が必要になります。シフト演算は 15 ビット以上で行われなければなりません。そのために、以下のように十分な精度を持つ左辺を準備します。

```
{spill,data_out} = data_in << shift;
```

左辺は 16 ビットなのでシフト演算は 16 ビットで行われる筈ですが、処理系が完璧でなければ、8 ビットのシフト演算になります。その場合、spill は常に 8'b0 になります。そのようなリスクを避けるためには、以下のように完全な記述を与える必要があります。

```
{spill,data_out} = {8'b0,data_in} << shift;
```

この記述によればシフト演算は 16 ビットで行われます。更に、以下のようにスケーラブルな記述をする機会につながります。

```
module shifter #(NBITS=8) (input [NBITS-1:0] data_in,...
...
{spill,data_out} = {{NBITS{1'b0}},data_in} << shift;
...
endmodule
```

このようにすると、shifter は 64 ビットでも 128 ビットでも動作します。

47　シンタックスの重要性　2022.05.22

SystemVerilog では、多少の違和感があっても文法的に正しい記述が多くあります。SystemVerilog の機能を最大限に活用するためには、LRM のシンタックスを理解するのが最善の策です。

LRM のシンタックスに注意を払う価値がある例を示します。

最も典型的な例は、以下のモジュール宣言だと思います。import 文をモジュール名称の直後におけるのは意外かも知れませんが、必要な機能です。

```
module universal_shift_register import pkg::*; #(NBITS=2)
    (input clk,reset,c1,c0,left_in,right_in,[NBITS-1:0] data_in,
    output logic [NBITS-1:0] q);
```

また、下記の例は文法的に正しい記述です。論理合成可能ではありませんが、記述可能です。実は、このような記述例は LRM に見当たりません。

```
if( state == S0 )
    #10 x = 0;
else if( state == S1 )
    #20 x = 1;
else if( state == S2 )
    #30 x = 2;
else
    #40 x = 'x;
```

(a)

```
case (state)
S0: #10 x = 0;
S1: #20 x = 1;
S2: #30 x = 2;
endcase
```

(b)

LRM を調べると文は以下のように定義されています。

```
statement ::= [ block_identifier : ] { attribute_instance } statement_item
statement_item ::= blocking_assignment ;
...
| nonblocking_assignment ;
| procedural_timing_control_statement
...
```

procedural_timing_control_statement が疑問に答えてくれます。

```
procedural_timing_control_statement ::= procedural_timing_control statement_or_null
...
procedural_timing_control ::= delay_control | event_control | cycle_delay
```

このシンタックスから、(a) と (b) の記述例は、文法的に正しい事になります。

48　パッケージの import　2022.05.23

SystemVerilog のパッケージの内容を簡便に参照するために import を使用しますが、グローバルスコープに import する事は避けた方が良いです。特に、UVM を使用する際には、このルールは鉄則と言えます。

モジュールヘッダでパッケージの内容を参照している場合、以下のようにモジュール名称の直後でパッケージを import しなければなりません。ここで、WIDTH と OPERATOR_WIDTH は Package に定義されています。

```
module ArithLogicUnit import Package::*; #(N=WIDTH)
    (input [N-1:0] A,B,[OPERATOR_WIDTH-1:0] Select,
    output CompareOut,[N-1:0] DataOut);
```

こうする代わりに、以下のようにすれば良いと推奨する人がいるかも知れませんが、好ましい方法ではありません。

```
import Package::*;
module ArithLogicUnit #(N=WIDTH)
    (input [N-1:0] A,B,[OPERATOR_WIDTH-1:0] Select,
    output CompareOut,[N-1:0] DataOut);
```

確かに、この一部だけ見れば、この対策でも良いのですが、パッケージ Package の内容がグローバルに見えてしまう危険性が問題なのです。つまり、Package 内に定義されている名称と同じ名称をパッケージ外で他の目的で再定義すると問題が発生します。

参考
モジュール名称の直後に import を指定できる機能は、前仕様 IEEE Std 1800-2012 で初めて導入されました。それ以前の仕様の LRM には記載されていないので、古い参考書をお読みの方は最新の LRM（IEEE Std 1800-2017）で確認する必要があります。多くの専門家が必要な機能であると認識しているので、グローバルスコープにパッケージを import しないルールを設けるのが最善の策です。

49　モジュールポートの種類　2022.05.27

SystemVerilog では、ネットと変数の差異はモジュールポートにおいて最も顕著に現れます。output ポートが最も難解ですが、ルールなので覚えるしかありません。

以下の説明はネットと変数の差異を明確にしていますが、データタイプ `logic` を使用せずに解説しています。つまり、`logic` が変数を宣言するための手段ではない事が分かります。

SystemVerilog 豆知識

機能	タイプ	仕様と注意点
モジュールポート	input	• input ポートは入力専用であれば何処でも使用できるので、ネットであろうと変数であろうと構いません。
	inout	• 最初のポートの方向を省略すると、inout が仮定されます。 • inout ポートは、ネットでなければなりません。つまり、キーワード var を指定できません。var さえ指定しなければ問題は起こりません。
	output	• ネットでも変数でも構いません。 • 厳密に言えば、データタイプが省略されるか、または `implicit_data_type` が指定されれば、ポートはネットになります。 • 逆に言えば、データタイプを指定すると output ポートは、変数になります。 • 尚、ポート名称以外の指定がない場合、直前のポートのデータタイプが仮定される事に注意して下さい。 • 従って、output ポートを変数にするためには、output とデータタイプを同時に指定すれば間違いは起こりません。
	ref	• ref タイプのポートは、変数でなければなりません。
非ポート	ネット	• ネット型のキーワードを指定して宣言するとネットになります。
	変数	• ネット型のキーワードを省略してデータタイプを指定すれば、変数になります。 • キーワード var を明示的に指定しても良いです。

`implicit_data_type` は以下のシンタックスを持ちます。

```
implicit_data_type ::= [ signing ] { packed_dimension }
signing ::= signed | unsigned
```

具体例は下記の文献にあります。
SystemVerilog LRM、704 ページ
SystemVerilog 入門、共立出版、273 ページ

50　import　2022.05.31

【まとめ】パッケージをグローバルに import しなくても、名称に関する問題が発生する事があります。その場合の問題解決には時間を要する事もあります。

LRM の例に加筆して解説します。先ず、簡単なパッケージを以下のように定義しておきます。

```
package pkg;
    int     x;
endpackage
```

このパッケージをモジュールスコープ内で import し、パッケージ内に宣言されている変数 x を二つの異なる場所で再定義します。最初の宣言は正しいのですが、二番目の宣言ではエラーになります。何故でしょうか。解説を読む前に自身で考えると良いです。

```
module test;
import pkg::*;

initial begin:a                          } ①
    x = 10;
end

initial begin:b
    int     x;      // ok                } ②
    x = 20;
end

int     x;          // illegal           } ③
endmodule
```

記述例の解説

記述コード	説明
①	モジュール内でまだ定義されていない変数 x を参照しているので、パッケージが検索されて x が見つかり、test.x==pkg::x の定義が確立します。
②	begin ブロック b 内には x が定義されていないので、変数 test.b.x が定義されます。
③	test スコープで変数 x を定義しようとしていますが、①により、既に test.x は pkg::x と定義されているため、エラーになります。

参考
パッケージ内で定義されている名称をパッケージ外で再定義しなければ、殆どの問題は起こりません。名称の重複を避けるために、パッケージ内で定義されている名称には uvm_のように一定の prefix を添えるのが一般的な手法です。

51 implicit_data_type 2022.06.04

SystemVerilog の implicit_data_type とは、いったい何者であるかを調べてみましょう。

結論から言えば、implicit_data_type はデータタイプの指定ではないのですが、この指定をするとデータタイプの指定ができなくなり、結果として、標準的なデータタイプの仮定が余儀なくされます。この意味において、暗黙のデータタイプ宣言機能と言えます。先ず、変数を定義する際のデータタイプ宣言のシンタックスは以下のようになります。

```
var_data_type ::= data_type | var data_type_or_implicit
```

data_type は、以下のシンタックスを持ちます。このシンタックスによれば、データタイプのキーワードを、必ず、先頭に指定しなければなりません。

```
data_type ::=
  integer_vector_type [ signing ] { packed_dimension }
  | integer_atom_type [ signing ]
  | non_integer_type
  ...
```

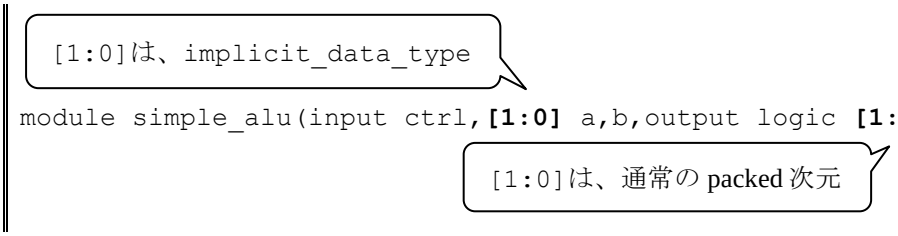
次に、data_type_or_implicit は以下のようなシンタックスを持ちます。問題の implicit_data_type が出てきました。データタイプキーワードの指定から始まれば、data_type_or_implicit は data_type のシンタックスとなり、データタイプキーワードを省略すると、data_type_or_implicit は implicit_data_type のシンタックスを選択する事になります。

```
data_type_or_implicit ::= data_type | implicit_data_type
```

implicit_data_type は以下のようなシンタックスを持ちます。しかも、このシンタックスにはデータタイプを指定する機能はありません。

```
implicit_data_type ::= [ signing ] { packed_dimension }
signing ::= signed | unsigned
```

以上から、implicit_data_type のシンタックスが選択されると、データタイプの指定がされていないため、標準的なデータタイプを仮定しなければなりません。それが、データタイプ logic の役目になります。最後に、例を示しておきます。



52　{} オペレータと繰り返し　2022.06.07

SystemVerilog の {} オペレータ内に繰り返しを指定した場合、繰り返しの対象となるオペランドは、一回しか評価されません。例えば、`{4{f(a)}}`において、`f(a)`は一回しか呼び出されません。

SystemVerilog の繰り返しは、同じ値の繰り返しを意味するので、繰り返しのオペランドは一回しか評価されません。例えば、以下のような繰り返しが与えられたと仮定します。

```
r = {4{f(a)}};
```

SystemVerilog では、この代入文は以下のように展開されます。

```
v = f(a);
r = {v,v,v,v};
```

更に注意すべき事は、繰り返しが 0 でも、評価が一回行われる点です。例えば、以下の繰り返しは 0 ですが、`f(a)`は一回呼び出されてしまいます。

```
r = {0{f(a)},8'hef};
```

参考
繰り返しのオペランドは一回しか評価されないので、以下の効果の違いに注意して下さい。

繰り返しを使用しない方法	繰り返しを使用する方法
`{$random,$random,$random}`	`{3{$random}}`
三つの 32 ビットの乱数を結合して 96 ビットの乱数を構成する。	一つの 32 ビットの乱数を合成して 96 ビットの数値を構成する。同じ値が三回続いてしまう。

53　SystemVerilog クラスの活用　2022.06.08

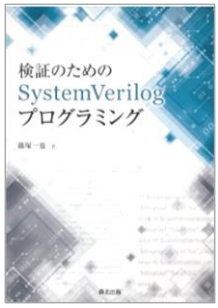
UVM を適用するほどの大規模な検証を意図しない場合には、汎用的な検証コンポーネントを SystemVerilog クラスとして準備しておくだけでも効果的です。

検証環境は、一般に、階層構造を形成してトップダウンに構築されます。したがって、階層構造を構成するために必要な機能を提供する検証コンポーネントのベースクラスを開発しておくと、検証環境構築の開発手順が簡略化されて作業効率が良くなります。つまり、毎回同じような手順を踏む必要がなくなります。

階層を構築するためには、UVM の uvm_component クラスと同じように、インスタンス名と階層の親コンポーネントを指定するだけで十分です。その他には、階層を操作するためのファンクションを準備しておけば、殆ど全ての小規模検証環境で再利用可能なベースクラスとなります。そして、必要に応じて、そのベースクラスに機能を追加して行けば、独自の検証ライブラリーの基を築けます。

```
class component_t;
string    name;
...
extern function new(string name,component_t parent);
extern function int add_child(component_t child);
extern function component_t get_parent();
extern function component_t get_child(string name);
extern function int get_first_child(ref string name);
extern function int get_last_child(ref string name);
extern function int get_next_child(ref string name);
extern function int get_prev_child(ref string name);
extern function string get_name();
extern function string get_full_name();
endclass
```

そのようなベースクラスの典型的な例は、component_t クラスとして下記の文献にあります。サブコンポーネントを管理する一般的な手法も同時に学べます。その他、ログファイルを見易くするための機能が GCL パッケージとして紹介されています。これらを活用すれば、小規模な検証では、UVM に依存しなくても良い機会が増えてきます。店頭で立ち読みすると良いです。



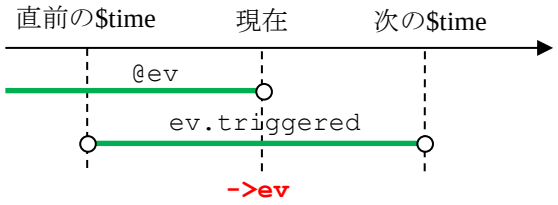
参考文献
検証のための SystemVerilog プログラミング、森北出版 、2022、
第 7 章、第 11 章

54　SystemVerilog 特有のイベント制御　2022.06.11

SystemVerilog 特有のイベント制御法を復習しておきます。

SystemVerilog 豆知識

使用例	仕様と注意点

`event ev;`
`@ev;`
`wait(ev.triggered);`
	• `ev` は値を持たないため、値による判定は不可能です。確実に言える事は、解除（`->ev`）が行われる前に、待ち（`@ev`）が有効になっていなければなりません。
	• `triggered` メソッドを使用すれば、解除とのタイミング問題を解消できますが、時刻が進むと `triggered` メソッドの状態はクリアされてしまうため、このメソッドが有効なのは僅かの時間です。
	• 下図は、イベント待ちが有効な区間を示していますが、白丸は、その時刻を含んでいない事を意味します。
	 
	
	• 詳細は、下記文献にもあります。後者の文献は、プロセスの実行順序依存性を詳しく解説しています。
	SystemVerilog による検証の基礎、森北出版、第 2.2.7 項.
	検証のための SystemVerilog プログラミング、森北出版 、第 3 章.

`semaphore lock;`
`lock = new;`
`lock.get(1);`
	• 実は、semaphore でもレベルセンシティブなイベント制御をできます。しかも、この方法には、`@ev` や `ev.triggered` が持つ時間的な制約がありません。
	• 興味のある方は、以下の文献を参照下さい。
	検証のための SystemVerilog プログラミング、森北出版 、第 4 章.

55 参考文献

文献[14]は、SystemVerilog の入門書を読んだ後、学習した知識を習得しているかを確認するために最適な書物で、進路決定に役立ちます。

市販されている書物を読んでも、どうしても SystemVerilog を理解し難いと感じている方には文献[15]をすすめます。この資料を根気よく読み続けると、自然に、SystemVerilog の理解へと導かれます。

[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] Ashok B. Mehta: SystemVerilog Assertions and Functional Coverage, Springer 2014.
[5] Ben Cohen, Srinivasan Venkataramanan, Ajeetha Kumari, and Lisa Piper: SystemVerilog Assertions Handbook, 4th Edition, VhdlCohen Publishing, 2016.
[6] Kathleen A. Meade and Sharon Rosenberg: A Practical Guide to Adopting the Universal Verification Methodology (UVM), 2nd Edition Cadence Design Systems, Inc. 2013.
[7] Stuart Sutherland, Simon Davidmann, and Peter Flake: SystemVerilog for Design, 2nd Edition, Springer 2006.
[8] 篠塚一也、SystemVerilog による検証の基礎、森北出版 2020.
[9] 篠塚一也、SystemVerilog 入門、共立出版 2020.
[10] 篠塚一也、実践 UVM 入門、森北出版 2021.
[11] Randy H. Katz: Contemporary Logic Design, The Benjamin/Cummings Publishing Company, Inc. 1994.
[12] Giovanni De Micheli, Synthesis and Optimization of Digital Circuits, McGraw-Hill 1994.
[13] F. Brown, Boolean Reasoning, Dover Publications, Inc. 2003.
[14] 篠塚一也、検証のための SystemVerilog プログラミング、森北出版 2022.
[15] 篠塚一也、SystemVerilog 超入門、アートグラフィックス 2021.