路の動作を保証できるようになります．新しい記法の多くは，従来，論理合成ツールに対する指示（プラグマ）として記述していた情報を，統一された言語仕様の中で記述できるようにしたものです．ツール間の取り扱いや解釈の差を解消し，設計者の意図と異なる回路が生成される状況を避けることができます．

3）注目されている機能検証手法に対応

SystemVerilogの最大の特徴とも言える点が，最近はやりの検証手法を取り込んだことです．前述のアサーション・ベース検証，制約付きランダム・テスト生成，機能カバレッジなどを，一つの言語の中で利用することができます．このほかにも，テストベンチを簡潔に記述するための機能などを備えています．

一つの言語の中でこれらの手法を利用できることにより，設計資産としての再利用性が向上し，また後述するように設計者と検証エンジニアの間のコミュニケーションが円滑になるといった効果が期待できます．

4）高抽象化記述

SystemVerilogは，記述の抽象度を上げる記法を用意しており，同時にC言語などで記述した機能モデルとの接続が容易になるという特徴もあります．

前者については，C言語と同じ2値型が広範囲に導入されています．これにより，とくにシミュレーションの高速化が期待できます．

後者については，DPI（Direct Programming Interface）と呼ばれる新しいインターフェースが用意されました．Verilog HDLでは，Cプログラムとのインターフェースとして PLIが提供されていますが，PLIには，記述が煩雑でシミュレーション実行時の速度オーバヘッドが大きいといった問題があります．DPIはこうした問題を解消します．Cモデルとの接続が容易になり，シミュレーション速度が向上することを期待できます．

● 設計者と検証エンジニアの言語を統一

SystemVerilogを利用することにより，システム・レベルのモデリングを除くほとんどの設計工程が一つの言語でカバーできることになります[注1-4]．これは設計チームにとって大きなメリットになります．

まず，言語習得については，移行期こそ習得のための相応の努力が必要となりますが，複数の言語を習得するよりははるかに楽になると期待されます．さまつなことですが，セミコロンなどの記号の使いかた一つとってみても，言語ごとに異なるスタイルを覚えることは非常に煩雑に感じます（そのうえ，これは設計業務の本質とは関係ない）．

プロジェクトを計画・管理する立場から言うと，利用できるツールの選択肢が増え，自由度が増すことにより，計画・管理が行いやすくなると期待できます．もっともこれは，課題として後述するように，EDAベンダのきちんとしたサポートが必要条件となります．

また，言語仕様として定義されたことにより，ツールに依存する制約が減少し，設計資産の活用がより円滑になると期待されます．

さらに，設計者と検証エンジニアが同じ言語を使用することにより，コミュニケーションが円滑にな

注1-4：その名称ゆえに誤解されがちだが，SystemVerilogではシステム・レベル・モデリングのための機能はそれほど強化されていない．一方，SystemCはシステム・レベル・モデリング向きの言語といえる．

ると期待できます．日本ではそれほど一般的ではありませんが，米国などの大規模な設計プロジェクトでは，与えられた仕様から回路を設計する設計者と，その回路を検証する検証エンジニアが分かれています．両者の職務を明確に区別し，異なる視点でクロスチェックを行うことにより，検証漏れが生じたり，市場に不良品が流れてしまうリスクを低減できると考えられています．

従来，設計者はもっぱらHDLのみを使用し，両者のインターフェースとなる仕様は不完全なフローチャートや自然言語で与えられることがほとんどでした．SystemVerilogを用いると，仕様をアサーションとして記述したり，設計者が設計の意図を明確に表現しやすくなります．これによりプロジェクト全体の設計生産性が向上することを期待できます．

● 設計記述についてはチェッカの整備が必要

SystemVerilogを導入・利用する際には，いくつか気をつけなければならない項目があります．

まず，現在Verilog HDLを入力としている各種EDAツールがSystemVerilogの必要な機能を完全にサポートするまでは，おもに設計（回路）記述において問題が生じる可能性があります．例えばシミュレータや論理合成ツールがSystemVerilogのある機能をサポートしていたとしても，そのほかの設計ツール（社内製ツールを含めて）がサポートしていないと，下流の設計工程で不ぐあいが生じる可能性があります．このように問題のある機能は，設計記述中の構文や構造を静的に解析する“チェッカ”と呼ばれるツールによって完全に排除する必要があります．つまり，社内で利用するツール全般のSystemVerilogのサポート状況を調査し，チェッカを整備することが必要となります．

ちなみに検証記述については，再利用性や移植性について留意すべき点はあるものの，上述のような設計上の致命的な問題はほとんどありません．SystemVerilogを使えるところから使っていく，という姿勢でよいと考えています．

もう一つ注意しなければならないのは，言語仕様に関するサポート状況を調べるだけでは，実用上問題がないのかどうかわからない機能が存在する点です．これはツール間の性能差と言語仕様の解釈の違いという形で現れる可能性があります．現状でも，例えば論理合成において，ツール間で能力差が顕著に現れます（生成される回路に優劣がある）．SystemVerilogでは，各ベンダとも新しい言語をサポートするということで，当初は実装の違いにより無視できない能力差が現れることが予想されます．代表的なものは，制約付きランダム・テスト生成です．ツール側のアルゴリズムによっては，パターン生成の効率に大きな差が生じる可能性があります．

言語仕様の解釈については，現時点で具体的にどのような問題が起こるのかを例示することは困難です．実際に言語仕様を策定しているのは限られた人数のエンジニアであるのに対して，今後，多数のEDAベンダやEDAユーザが評価にかかわってきます．そうなると，言語仕様上のあいまいな点が浮き彫りになってくる可能性があります．

＊　　＊　　＊

Verilog HDL 2001が登場したとき，「使ってみたい」と思いながら，現実にはそうならず，失望したことがありました．仕様の策定にEDAベンダの代表が含まれていなかったことがその一因と言われています．一方，SystemVerilogについては，大手EDAベンダの代表が主導で仕様策定を推進してきた経緯があります．すべての機能が早期に利用可能となることを期待したいと思います．

第2章

記述能力，再利用性，検証機能を強化したSystemVerilog

赤星博輝

第2章ではSystemVerilogの特徴について解説する．SystemVerilogはVerilog HDL 2001（IEEE 1364-2001）に続くVerilog HDLの言語仕様である．すでに，多くのEDAベンダがサポートを表明している．SystemVerilogでは，記述量を減らしたり，記述ミスを減らすための構文が追加されている．また，通信方式の再利用に有効なインターフェースの概念に対応している．さらに，アサーションやランダム・テスト生成など，検証のための機能も新たに用意された．（編集部）

ハードウェア記述言語を含む設計言語の進化は，設計する回路の大規模化および複雑化が一つの要因となっています．米国Intel社の創設者のひとりであるGordon E. Moore氏が「LSIに集積されるトランジスタ数は，2年で倍のペースで増える」と述べたムーアの法則で示されるように，LSIで実現できる回路規模は指数関数的に増加してきました．回路規模が大きくなることにより，その大規模な回路を効率良く設計する手段や設計検証項目が増大することへの対応が必要となります．また，事前の性能評価などの検討がより重要となってきました．

Cベース設計では，C言語やC＋＋などのデータ型を使ってシミュレーションすることで，検証速度を高速化しています．また，ビヘイビア合成を使うことにより，RTL設計より記述量を削減できるなどの利点もあります．さらに，最近のLSIでは機能をソフトウェアとして実現する比率が高く，ソフトウェア開発とリンクしやすいということもありますし，ハードウェアとソフトウェアを合わせたシステムとして包括的に設計を行える可能性もあります．

● 記述量を削減し，モデリングと検証のための機能を強化

それではなぜ，わざわざSystemVerilogが開発されたのでしょうか？ これは，ハードウェアの設計ではVerilog HDLが主要な言語の一つであるからです．これまでの設計資産や設計手法を捨てて，新しい設計言語や新しい設計手法に移行することはそれほど簡単ではありません．ただし，既存のVerilog HDLにはいくつかの問題点がありますし，最近登場した言語と比べると機能的に遅れている点もあります．そのため，これまでのVerilog HDLの設計資産や設計手法をそのまま使用でき，さらに新しい機能が使えるようになったSystemVerilogが注目されています．

SystemVerilogの新しい機能としては，以下の点を挙げることができます．

- RTL記述をより効率的に行える
- 高位のモデリングを容易に行える

●検証向けの機能を言語仕様として用意している

設計を効率良く行う（端的に言えば，記述量を減らす）ためには，高位のビヘイビア・レベル記述を使う手法があります（ビヘイビア・レベル記述による設計はあまり普及していないが…）．これに対して，SystemVerilogではVerilog HDL言語を改良することで，RTLでも記述量を減らせるようになりました．これまでの設計手法を使用したまま記述量を減らせることは，設計者にとって利点となります．

また，これまでは高位のモデリングはC言語やC＋＋が中心的な役割を担っていましたが，SystemVerilogではC言語とのリンクも含め，高位のモデリングが行いやすくなりました．

検証についてはVerilog HDLのサポートが弱く，最近，検証のための専用言語（いわゆる検証言語）などを使用する事例が増えています．SystemVerilogでは検証に対する機能を大幅に追加し，別の言語を用いる必要がなくなりました．

ここでは，Verilog HDLの問題点や機能的に遅れている点を中心に，SystemVerilogで大きく変わった点について解説していきます．

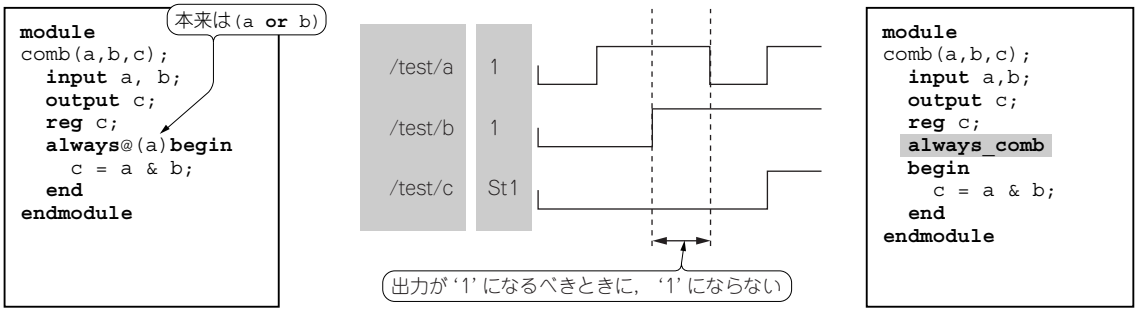
● always文を大幅に改良

まずはalways文の問題から見ていきます．always文には設計時にミスが発生しやすいという問題がありました．

1）always_comb文の追加

always文を使って組み合わせ回路を記述する場合，センシティビティ・リストに必要な信号をもれなく記述しなければなりません．例えば，**図2-1**の例では2入力ANDを作成しようとしたのですが，センシティビティ・リストが不十分であるため，シミュレーションは誤った結果になっています．しかし，よく考えてみるとセンシティビティ・リストを書くという作業は，設計者のためというよりも，昔，このように書くと決められたからというものです．

この問題を解決するため，SystemVerilogではalways_combという文が追加されています．これにより，設計者はセンシティビティ・リストを書く必要がなくなります．これは，設計者にとってはうれしいことです．複雑な回路を記述していると，どうしてもセンシティビティ・リストに書く信号が多く



（a）まちがった記述　（b）シミュレーション結果　（c）always_combを使った記述

図2-1　まちがったセンシティビティ・リスト
(a)のようなまちがったセンシティビティ・リストでシミュレーションを行うと，(b)のように入力aとbが‘1’のときに出力cが‘1’にならず，‘0’になったりする．こういうミスは急いでいるときほど発生しやすい．SystemVerilogの場合，組み合わせ回路の記述にalways_combを使えば，(c)のようにセンシティビティ・リストは不要である．

リスト2-1　always_latchを使った記述
これまでの記述と異なり，センシティビティ・リストは不要になる．回路構造を明示的に指定できることで，デバッグ・ツールなどの支援を期待できる．

```
module case1(a,sel,c);
  input a;
  input[1:0]sel;
  output c;
  reg c;

  always_latch

  case(sel)
   2'b00:  c = a;
   2'b01:  c = 1;
   2'b10:  c = 0;
  endcase

endmodule
```

これまでの記述では
always@(a or sel)

なるため，ミスしやすくなります．always_comb文を使えばセンシティビティ・リストに関するミスがなくなり，センシティビティ・リストが不要になるため，記述量も減ることになります．

2）always_latch文の追加

組み合わせ回路を記述するためにalways文の中でif文やcase文を使うときに，すべての条件が記述されていないとラッチを生成するという問題があります．もちろん，ラッチを記述したい場合はそれで良いのですが，条件のもれによって発生してしまったというケースが多いようです．

そこで，SystemVerilogではラッチを明示的に記述するためにalways_latch文が追加されました（**リスト2-1**）．これにより，ラッチを記述する場合にはalways_latchを使用し，組み合わせ回路を記述する場合には先ほどのalways_combを使用することで，これまでのようなミスを減らすことができます．さらに，ツールから出力されるメッセージもより的確になることが期待されます．

3）always_ff文の追加

組み合わせ回路とラッチが明示的に指定できるようになったら，次はフリップフロップです．SystemVerilogではalways_ff文が追加されました．フリップフロップの記述については，これまでのalways文の代わりにalways_ffを使う積極的な理由が，筆者にはちょっと思い浮かびません．しかし，回路構造を明確に記述するという観点からは，今後はalways_ffを使った記述に変わっていくのではないかと思います．

● case文のアトリビュートの問題を解消

これまでfull_caseやparallel_caseといったアトリビュートを使用された方も多いかと思いますが，実はこのアトリビュートは問題を発生させる要因となることがあります．RTLシミュレーションと論理合成後のゲート・レベル・シミュレーションの間で結果が異なる可能性があることが知られています．

リスト2-2に示すように，SystemVerilogではif文やcase文で使用可能なuniqueとpriorityの二つのキーワードが追加されました．priority caseでは最初に一致した条件に対する処理のみ実行されますし，unique caseではすべての条件で重なりがないことを設計者が保証することになります．uniqueはpriorityと異なり，並列に評価することが可能となります．また，uniqueやpriorityをcase文で使用する場合，defaultの条件が不要ならばdefaultの記述も不要となります．

リスト2-2　uniqueとpriorityを使った例
(a)のようにuniqueを使うことで，条件に重なりがないことを指定できる．また，(b)のようにpriorityを使うことで先頭から評価することを指定できる．アトリビュートによる指定からキーワードに変わったことで，今後，合成結果とのシミュレーションの不一致がなくなることを期待したい．

```
unique case(a)
  0,1:  $display("0 or 1");
  2:    $display("2");
  4:    $display("4");
endcase
```

（**a**）条件に重なりがない例

```
priority casez(a)
  3'b00?: $display("0 or 1");
  3'b0??: $display("2 or 3");
  default: $display("4 to 7");
endcase
```

（**b**）記述された順番に評価する例

リスト2-3
ポート宣言とそのデータ型の定義
SystemVerilogやVerilog HDL 2001では，まとめて定義することができる．これにより，ポートを定義する場合のミスを減らせる．

```
module  mux8(y,a,b,en);
  output[7:0] y;
  input[7:0]  a,b;
  input       en;

  reg[7:0]    y;
  wire[7:0]   a,b;
  wire        en;
  .....
```

（**a**）Verilog HDLによる記述

```
module  mux8(
  output reg[7:0]  y,
  input  wire[7:0] a,
  input  wire[7:0] b,
  input  wire      en);
  .....
```

（**b**）SystemVerilogによる記述

リスト2-4
センシティビティ・リストの区切り
SystemVerilogやVerilog HDL 2001では，orの代わりにコンマ(,)で区切ることができる．orを使うと，一瞬，信号の指定かと思ってしまうことがある．ちょっとしたことだが，可読性が上がるので，SystemVerilogの利点の一つと言ってよいのではないだろうか．

```
always@(sel or
        a or b or c or d)
case (sel)
  2'b00: y = a;
  2'b01: y = b;
  2'b10: y = c;
  2'b11: y = d;
endcase
```

（**a**）Verilog HDLによる記述

```
always@(sel,a,b,c,d)
case(sel)
  2'b00: y = a;
  2'b01: y = b;
  2'b10: y = c;
  2'b11: y = d;
endcase
```

（**b**）SystemVerilogによる記述

● 冗長な記述が不要に

Verilog HDLでは，記述量が長くなってしまう部分があります．すでに説明したalways_combなどでは，センシティビティ・リストが不要になりましたが，ほかにもいくつか改善された点があります．

1）ポートとその型の同時宣言

Verilog HDLのポートとその型の宣言などは，数ヵ所に書く必要があります．これは煩雑な作業であり，ツールに読み込ませるときにどこかが不足しているなどの理由でエラーとなったことのある方もいらっしゃるでしょう．

リスト2-3に示すように，SystemVerilogでは1ヵ所でまとめてポートに関する定義を行えるようになりました．Verilog HDLの記述と比べてみると，コンパクトに記述できることがわかります．

2）センシティビティ・リストの区切り

always文ではセンシティビティ・リストを書きますが，大きなセレクタなどの場合，多くの信号を定義する必要があります．Verilog HDLではorで区切りますが，Verilog HDL 2001やSystemVerilogではコンマ(,)で区切ることもできます(**リスト2-4**)．個人的には，orで区切ると信号名と同じアルファベットで記述されているので，やや可読性が低く感じます．一方，コンマで区切ると信号名とはっきり区別できます．ただし，SystemVerilogではalways_combやalways_latchが導入されたため，

有効性を感じる場面は少ないかもしれません．

3）1ビット信号の展開

SystemVerilogでは，ビット幅指定のない‘0’，‘1’，‘x’，‘z’は自動的に展開される機能が追加されています．ビット幅が大きい信号に対して`all 0`や`all 1`などを代入したい場合は，便利な記述方法だと思います．

● 階層に関する機能を多数追加

SystemVerilogでは階層に関する記述についても，記述量を減らすための機能や再利用のための機能などが追加されています．代表的なものを説明します．

1）ポート記述の.*

Verilog HDLでは，下位階層モジュールはインスタンスして使用していましたが，そのときにポートに対する接続方法としては，定義された順番で接続する方法と名まえによって明示的に接続する方法がありました．しかし，ポート数が多いと，どちらの方法でも記述がたいへんです．**図2-2**に示すように，SystemVerilogでは.*と書くことで接続する信号を自動的に推定してくれます．

また，.*と書く方法と名まえによって明示的に接続する方法を混在して使用することもできます．例えば，クロックやリセットのみを.*によって接続するなどが考えられます．ポートの接続を直接指定した場合，.*による接続よりも直接記述されたほうが優先されます．

2）トップ・レベル（$root）の定義

SystemVerilogでは，`task`や`function`だけでなく，変数もモジュール内に定義します．このため，デバッグなどで共通に使いたいものは，**リスト2-5**のようにトップ・レベルに記述します．トップ・レベルにあるものはどの階層でも使用できます．これによって，デバッグや解析を効率良く行うことができます．

3）ネストしたモジュール

Verilog HDLでは，モジュールの中にモジュールを持つことはできませんでした．SystemVerilogではモジュールの中にモジュールを持つことができます．一つは大きなモジュールを論理的に分割するため，もう一つは下位モジュールを定義するためという二つの使いかたが想定されます．

大きなモジュールを論理的に分割した例を**リスト2-6**に示します．ただモジュールを分けるだけなら，あまり利点はないように思われるかもしれません．しかし，論理的な区切りをつけるためにポートを使用しなくてよいという特徴があります．また，下位階層の中で信号を定義できるので，トップ階層で定義する信号を減らせる場合もあります．

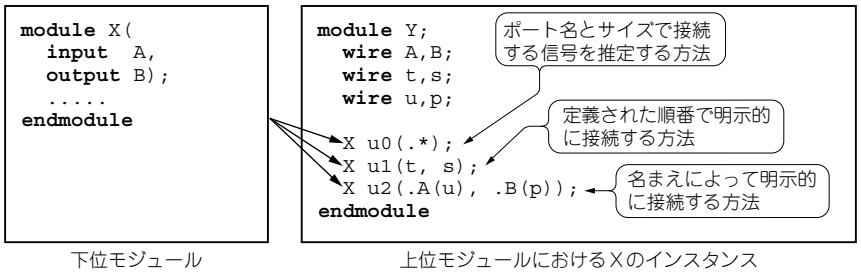


図2-2　ポート記述の.*
インスタンス時にポートに.*と書くことで，ポート名とサイズを見て接続する信号をツールが推定してくれる．

リスト2-5　トップ・レベル（$root）の定義
Task，functionだけでなく変数も定義できる．これにより，デバッグ時にポートを使用しなくても，下位階層にデータを渡すことが可能となる．

```
/*  ファイルの先頭    */
typedef int myint;

myint counter0;

function void incr( output myint   k );
   k = $root.counter++;;
   $display("entering left");
endfunction
.....
```

リスト2-6　論理的な分割のためにネストしたモジュール
論理的な分割を行うために，モジュールの中にモジュールを定義した．ポート定義が不要な点が特徴である．

```
module dff_nested(
    input d, ck, pr, clr,
    output q, nq);
  wire q1, nq1, nq2;

  module ff1;  // module定義
    nand g1b (nq1, d, clr, q1);
    nand g1a (q1, ck, nq2, nq1);
  endmodule
  ff1 i1;  //インスタンス
  // ....
endmodule
```

（a）設計資産などではモジュール名の衝突が起こらないとは限らない

```
module part1(....);
  module and2(input a; input b; output z);
  ....
  endmodule
  module or2(input a; input b; output z);
  ....
  endmodule
  ....
  and2 u1(....), u2(....), u3(....);
  .....
endmodule
```

（b）ローカルなモジュールの記述例

図2-3　ローカルなモジュール定義
複数の異なったモジュールに同じ名まえが付けられることがある．異なるグループが開発した場合，このような状況を未然に防げないことも珍しくない．モジュール命名ルールで逃げるという方法もあるが，SystemVerilogのローカルなモジュール定義を利用するのも有効である．

モジュールの中で下位モジュールを定義できることは，これまでのVerilog HDLと大きく異なるところです．新規に設計を行う場合，異なったモジュールに同じ名まえが付くことはないと思います．しかし，過去の設計資産などを活用すると，異なったモジュールに同じ名まえが着けられていることがあります〔**図2-3**（**a**）〕．下位モジュールを内部で定義することで，再利用時にモジュール名が衝突したりすることがなくなるという利点があります．下位モジュールを定義する場合の例を**図2-3**（**b**）に示します．このような機能をうまく使うことで，再利用性を高めることができます．

● インターフェースの概念を導入

SystemVerilogの新機能の一つにインターフェースがあります．この機能はSystemCやSpecCといった言語にはすでに導入されています．このことが「Verilog HDLが遅れている」と筆者が感じていた一番の理由でした．

モジュール間の通信を記述する場合，これまではポートを定義し，モジュール本体に通信に関する処理を記述すると同時に，通信以外の処理も記述していました．

図2-4　インターフェースの利点
モジュール間の通信（インターフェース）では，ポート定義と通信用の記述を行う．これまでは，通信以外の処理の記述といっしょになってしまっていることが多く，通信方式の変更はかなりめんどうだった．インターフェースの概念を利用し，通信の処理を分離して記述すれば，通信方式の変更や再利用が容易になる．

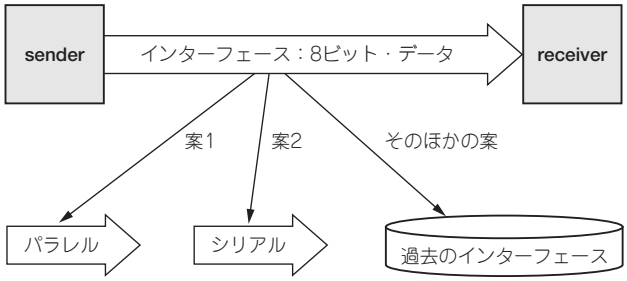


図2-5　インターフェースの交換
senderからreceiverに8ビット・データを送信するには，8ビットを一度に送るパラレル転送や1ビットずつ送るシリアル転送など，いくつかの方式がある．これらを簡単に交換できると，異なったインターフェースによる性能の違いをシミュレーションによって評価しやすくなる．インターフェースの概念を利用すると再利用性も向上するので，既存のインターフェースも容易に組み込むことができる．

一方，インターフェースを用いた設計では，通信に関する処理をインターフェースに実装し，モジュールには通信以外の処理を実装するようになります．通信の処理をインターフェースに詰め込んでいるので，設計者は使用するインターフェースを変更するだけで，異なった通信方式に対応できます（**図2-4**）．これにより，異なった通信方式を使ったシステムの性能評価が行いやすくなりました．

図2-5の8ビット・データを転送する例をもとに，実際にどのようにインターフェースを使うのかを説明していきます．

1）senderとreceiverの作成

すでにインターフェースがあって，**図2-6（a）**のようなタスクが存在する場合，モジュールを設計することができます．モジュールでは，使用するインターフェースを（必要な数だけ）ポート宣言します．

図2-6（b）はsenderの設計例です．1回だけデータを送るモジュールで，`interface A`のタスク

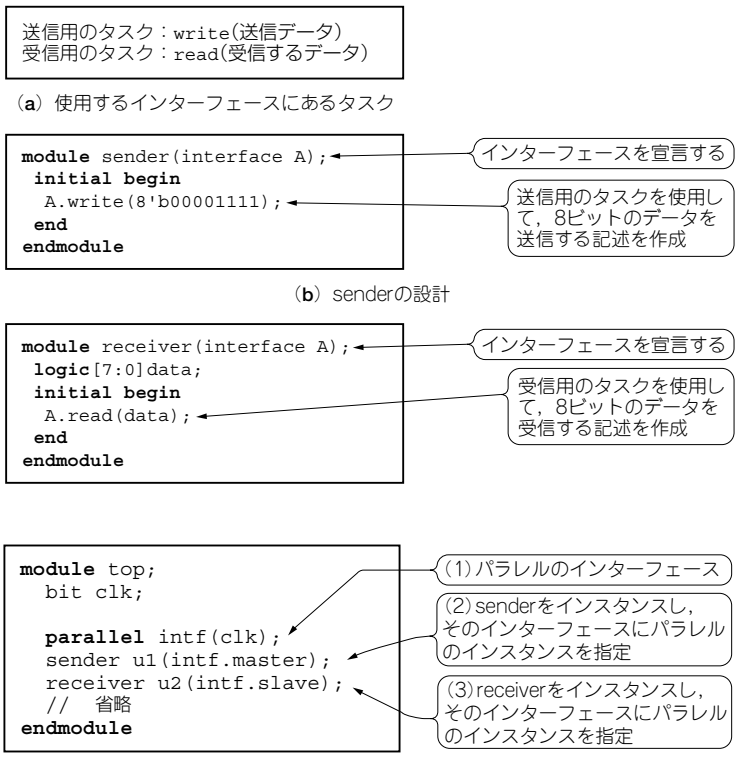


図2-6　インターフェースを用いたモジュールの作成
モジュールの設計では，使用するインターフェースをポート宣言し，用意されたタスクを使ってデータの送信や受信を行う．

リスト2-7　インターフェースの指定と交換
使用するインターフェースをインスタンスし，それを使用するモジュールで指定すれば，設計は完了である．シリアル・インターフェースにしたい場合は(1)で定義した部分を`serial intf(clk);`に変更すればよい．

```
module top;
  bit clk;

  parallel intf(clk);
  sender u1(intf.master);
  receiver u2(intf.slave);
  //   省略
endmodule
```

(1)パラレルのインターフェース
(2)senderをインスタンスし，そのインターフェースにパラレルのインスタンスを指定
(3)receiverをインスタンスし，そのインターフェースにパラレルのインスタンスを指定

writeを呼び出して，データ8'b00001111を送信しています．

図2-6(c)はreceiverの設計例です．1回だけデータを受け取るモジュールで，interface Aのタスクreadによってデータを受信します．

2）インターフェースの指定方法

次に，パラレルやシリアルなどのインターフェースを指定する方法を，**リスト2-7**の例で説明します．まず，使用するインターフェースをインスタンスします．この例ではパラレルについてインスタンスし，インターフェースで必要な信号を与えています．次に，そのインターフェースを使用するモジュールをインスタンスすればよいのですが，このときインターフェースのインスタンスを指定することで，使用するインターフェースを決定したことになります．

インターフェースをパラレルからシリアルに変更する場合は，**リスト2-7**のパラレルのインスタンス行を`serial intf(clk);`に変更するだけです(インスタンス名を変えるなどして，変更することも可能)．再利用が容易なので，設計グループの間で一度記述したインターフェースを共有するしくみを作っておくとよいでしょう．

3）インターフェースの作成

実際のインターフェースの例を**リスト2-8**に示します．最初にパラレルというインターフェースを宣言し，その後で，必要な変数などの定義を行います．この例ではsenderからreceiverの方向にデータを流すので，方向性があります．そのためmodportにおいてmasterの信号はoutputで定義し，

リスト2-8　パラレル・インターフェースの例
インターフェースで必要なwriteとreadの二つのタスクを準備している．基本的な動作については，Verilog HDLを知っていれば問題なく理解できるだろう．

```
interface parallel(input bit clk);
  logic[7:0]  data;
  logic       en=0;
  modport master(output data,
                 output en);
  modport slave(input data,
                input en);
  task write(input logic[7:0] d);
    data <= d;
    en   <= 1;
    @(posedge clk)
    data <= 0;
    en   <= 0;
  endtask

  task read(output logic[7:0] d );
    while (en ! = 1) @(posedge clk);
    d = data;
    @(posedge clk);
  endtask
endinterface
```

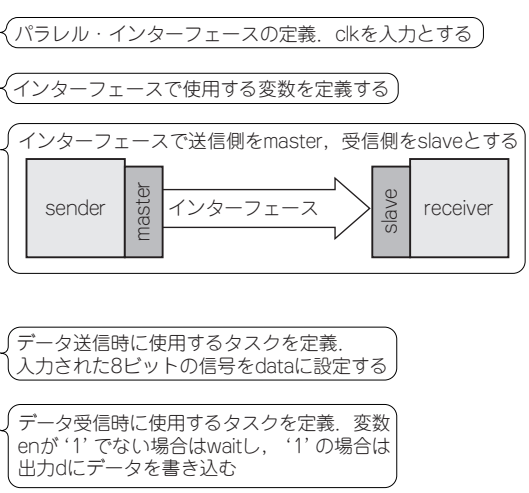


表2-1　SystemVerilogの2値データ型
4値データ型では‘0’，‘1’，‘X’，‘Z’を扱う必要があるのに対して，2値データ型では‘0’，‘1’だけを扱う．2値のデータ型を使うことで，シミュレーションが高速になり，メモリ使用量も減る．

データ型	機　能
shortint	2値，16ビット，符号付き整数
int	2値，32ビット，符号付き整数
longint	2値，64ビット，符号付き整数
byte	2値，8ビット，符号付き整数またはASCII
bit	2値，ユーザ指定サイズ，符号なし整数

slaveの信号はinputで定義します．**リスト2-7**のsenderとreceiverのインスタンスを行うときにintf.masterやintf.slaveと記述したのは，**リスト2-8**のmodportのmasterとslaveになるためです．

● さまざまな2値のデータ型を用意

最近のCベース設計の環境は，設計抽象度の引き上げによる検証速度の向上をねらっている場合が多いようです．一方，SystemVerilogではRTLをコンパクトに記述する機能や検証しやすくするための機能は追加されましたが，シミュレーション速度については，言語レベルではあまり改善されていないように見えます．

シミュレーション速度に関連する改善点としては，2値のデータ型を多く用意したことが挙げられます（**表2-1**）．4値のデータ型が不要な場所で2値のデータ型を使用すれば，シミュレーション速度は向上します．

● アサーションやランダム生成などの検証機能を標準装備

Verilog HDLには検証に対するサポートが弱いという問題があります．そのため，最近ではテストベンチの作成にC/C＋＋やe言語，Veraを使用する例が増えています．

条件を評価して
"pass(条件が真)"するとpass_statementを実行
"fail (条件が偽)"するとfail_statementを実行

```
assert(条件) pass_statement else fail_statement;
```

記述例

```
assert (foo) $display("pass"); else $display("fail");

assert (y==0) else flag = 1;
```

図 2-7　Immediate Assertion
チェックしたときの条件が成り立つか成り立たないかによって，処理を変更する．このようなアサーションを記述中に埋め込むことで，早い段階でエラー箇所を特定できるようになる．

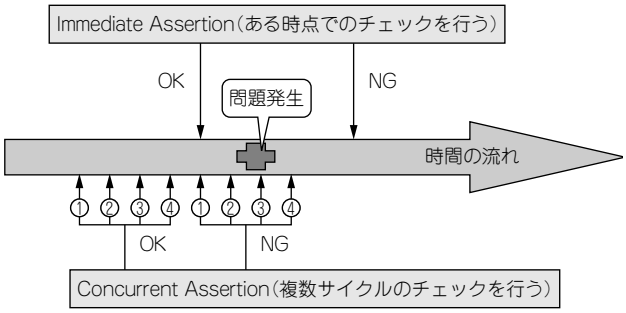


図 2-8　Immediate Assertion と Concurrent Assertion
Immediata Assertion では，ある時点の値だけで動作を判断するしかない．そのため，異常を検出しても，問題が発生したのはずっと前のサイクルである可能性がある．一方，Concurrent Assertion は複数サイクル(複数の時点，ここでは①，②，③，④)の値で判断する．そのため，Immediate Assertion の場合よりも問題箇所を特定しやすい．

SystemVerilog では，検証のための機能の多くを Vera の言語仕様から持ってきています．標準化されたことにより，今後は多くの EDA ベンダの検証ツールで SystemVerilog が利用できるようになると思われます．設計と検証を一つの言語で行えることは，設計者にとっては利点となります．

追加された検証機能について，順番に説明していきます．

1）二つのアサーション

SystemVerilog では即時アサーションと並列(コンカレント)アサーションという二つのアサーションが用意されました．即時アサーションは，その文が評価されたときに条件を満たしているかどうかをチェックするものです．これに対して並列アサーションは，複数サイクルにまたがった条件をチェックするものです．

図 2-7 は即時アサーションの記述例です．VHDL の assert 文と似ていますが，条件によって実行する文を切り替えることができます．回路中のさまざまな制約(例えば，「信号 X は 0 にならない」とか「信号 Y は 255 を超えない」など)を記述中に埋め込むことで，エラーが発生したときにエラー箇所を特定することが容易になります．エラーが発生したときは「何が原因で，どこで発生したのか」を突き止めることが最初の仕事になります．アサーションは，その解析を容易にしてくれます．ただし，この即時アサーションでは，ある時点の信号しか見ておらず，複数サイクルの動作をチェックすることはできません．

図 2-8 のように時間が経過しているシステムの場合，ある時点の信号だけを見て，回路の動作の正誤を判定することは容易ではありません．判断できたとしても，最初の問題が発生してから，かなりの時間が経過しているということが一般的だと思います．並列アサーションでは，複数のサイクルの動作をチェックすることができます．複数のサイクルでチェックする項目のことをプロパティと呼びます．記述例とプロパティを**図 2-9** に示します．

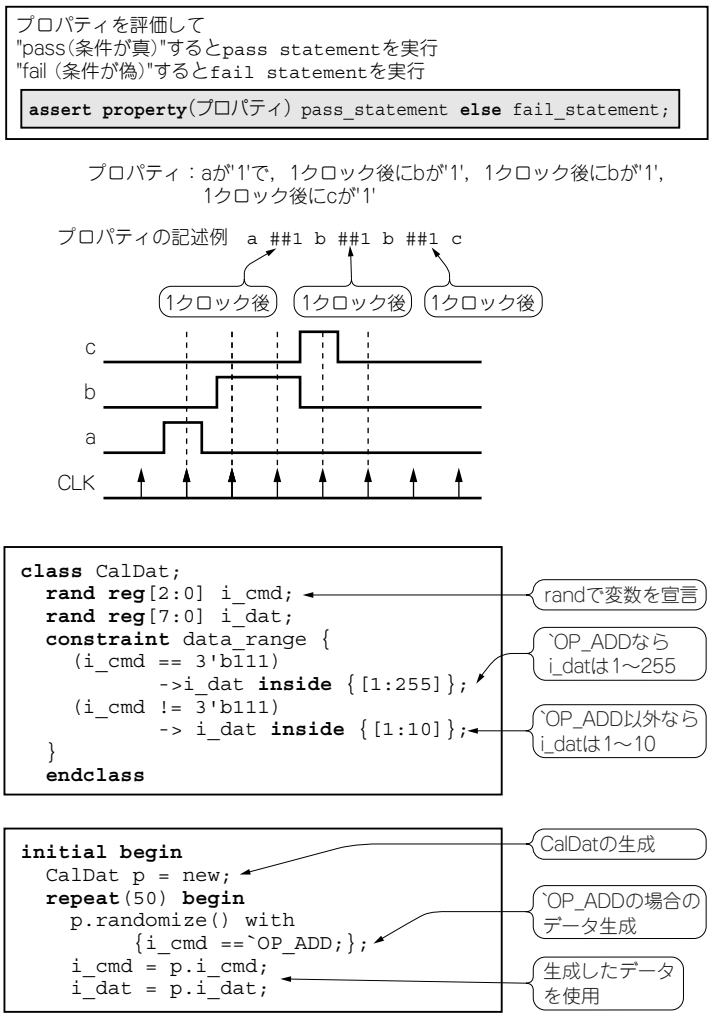


図2-9　並列アサーションの例
assert propertyで始まる記述が並列アサーションとなる．プロパティの真偽によって実行する文を選択する．プロパティの記述例では，タイミング・チャートの動きを指定している．

リスト2-9　ランダム・テスト生成
randをつけて宣言すると，シミュレーション・パターンをランダム生成することができる．constraintによって生成するパターンに制約を与えることで，むだなパターンをできるだけ生成しないようにする．

```
class CalDat;
  rand reg[2:0] i_cmd;
  rand reg[7:0] i_dat;
  constraint data_range {
    (i_cmd == 3'b111)
          ->i_dat inside {[1:255]};
    (i_cmd != 3'b111)
          -> i_dat inside {[1:10]};
  }
  endclass
```

randで変数を宣言
`OP_ADDなら i_datは1～255
`OP_ADD以外なら i_datは1～10

```
initial begin
  CalDat p = new;
  repeat(50) begin
    p.randomize() with
        {i_cmd ==`OP_ADD;};
    i_cmd = p.i_cmd;
    i_dat = p.i_dat;
```

CalDatの生成
`OP_ADDの場合のデータ生成
生成したデータを使用

プロパティはクロック単位の動作で記述します．Verilog HDLでは#で遅延を表現しましたが，SystemVerilogでは##でクロックを指定することになります．

図2-9の例ではタイミング・チャートで示している信号aが'1'になってから4サイクルの動作を定義しています．このプロパティを2回繰り返す場合は，[*2]を使うことで，a ##1 b[*2]##1 c のようにコンパクトに記述できます．

2）ランダム・テスト生成

SystemVerilogでは，**リスト2-9**のように変数にrandを付けることで，ランダムなシミュレーション・パターンを生成できます．中にはランダム生成と聞くだけで拒否反応を示す人もいるようですが，制約を与えながらランダムなパターンを生成することで，シミュレーションを効率化できます．また，アサーションとランダム・テスト生成を組み合わせて利用する方法も有効です．

3）サイクル・ベースのテストベンチ記述

これまでの Verilog HDL では，テストベンチは基本的には遅延時間を使って記述していました．SystemVerilog ではクロッキング（clocking）というしくみを導入し，クロックを基本としたテストベンチを作成します．クロッキングでは，クロックに対していつデータを出力するか（出力スキュー），また，いつデータを入力するか（入力スキュー）について，テストベンチとは別に指定することができます（**リスト 2-10**）．これにより，入力や出力のタイミングを変更することなどが容易になります．

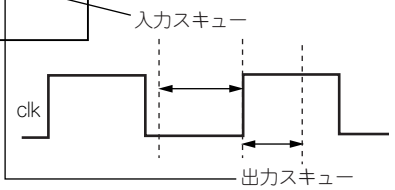
4）DPI（Direct Programming Interface）

C 言語の関数を SystemVerilog で取り扱ったり，SystemVerilog の関数を C プログラムの中で利用するためのしくみとして，DPI（Direct Programming Interface）が用意されています[注 2-1]．例えば，**リスト 2-11** のように C 言語の関数を SystemVerilog から呼び出すことが可能です（逆も可能）．System

リスト 2-10
clocking domain 定義とテストベンチ
clocking domain は，クロック（立ち上がり）に対していつデータを入力し，いつ出力するかを定義する．従来，テストベンチ記述では，# を使って遅延を定義していた．SystemVerilog では，## によってクロック数を指定することができる．

```
clocking A @(posedge clk);
  input  #3ns Q;
  output #6ns reset;
endclocking
```

（**a**）clocking domain 定義




```
initial begin
      A.reset <= 1;
  ##1 A.reset <= 0;
end
initial begin
  ##1 ;
  ##1 if (A.Q == 8'd0) $display("OK");
// ....
end
```

出力はクロック立ち上がりから6ns遅れでドライブ

入力はクロック立ち上がりの3ns前でサンプリング

（**b**）テストベンチ記述

リスト 2-11　DPI（Direct Programming Interface）
C 言語の関数を SystemVerilog で使用する場合，import 宣言を利用する．SystemVerilog の関数を C 言語で使用する場合は，SystemVerilog で export 宣言を行い，C 言語で extern 定義を行う．

```
module TEST;
  import "DPI" function void
    slave_write(input int adr, input
                              int dat);
  export "DPI" function write();

  function void write(int adr, int
                                  dat);
    slave_write(adr, dat);
  endfunction
  //....
endmodule
```

（**a**）SystemVerilog の記述

```
#include "svdpi.h"
extern void write(int, int);

void slave_write(const int I1, const
int I2)
{  /*  .....  */
}
```

（**b**）C 言語の記述

注 2-1：C 言語とのインターフェースという意味で，「DPI-C」と呼ぶこともある．

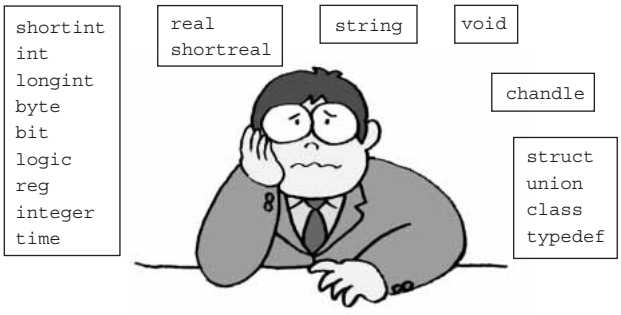


図2-10　SystemVerilogで使用できる代表的なデータ型
多くのデータ型がサポートされたことでモデリングが容易になり，C言語とのリンクも簡単に行える．これらのデータ型をどう使っていくかは，設計者の裁量に任されている．

Verilogでは**図2-10**に示すように，C/C++のデータ型が多く導入されました．DPIやさまざまなデータ型をいかに活用するかは，設計者の腕の見せどころとなるでしょう．

●「検証に対する機能」の導入は少し敷居が高い

SystemVerilogの新しい機能は，大きく以下の三つに分類することができます．

- RTL記述に関する機能
- モデリングを容易にする機能
- 検証に対する機能

「RTL記述に関する機能」は，これまでの設計手法を変えずに適用できます．また，設計上のミスが減り，記述量が減るという利点があります．

「モデリングを容易にする機能」のうち，筆者がいちばん注目したのはC言語の多くのデータ構造をサポートしたことです．これは，従来，C/C＋＋で作成していたモデルをSystemVerilogでも作成できるようになったと言い換えることもできます．また，DPIを使えば，C言語とSystemVerilogの間で関数をお互いに利用し合うことが可能です．`interface`文などの導入により，インターフェース（通信方式）を再利用しやすくなることも，モデリングを容易にすると思います．C言語とVerilog HDLの知識があれば，これらの機能は容易に導入できます．

「検証に対する機能」は効率的な検証を行ううえで重要な機能ですが，導入に対して少し敷居が高い機能になると思います．今後は，プロパティとランダム・テスト生成をうまく使いこなせるかどうかが，設計者，もしくは検証エンジニアにとって重要となってきます．

参考文献

(1) SystemVerilogのホームページ，http://www.systemverilog.org/