Artgraphics

SystemVerilog によるプログラミング

~ ハードウェア記述言語の検証分野への適用技術 ~

Document Identification Number: ARTG-TD-001-2021
Document Revision: 1.0, 2021.04.30
アートグラフィックス
篠塚一也

SystemVerilog によるプログラミング

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

SystemVerilog Programming

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

はじめに

SystemVerilog は、設計、仕様、検証機能を統一的に記述する事ができるハードウェア記述言語です。しかし、設計分野においては、RTL 論理合成のモデリングルールに従わなければならない規則により、使用できる SystemVerilog 機能は定型的な機能範囲に限定される場合が多いと考えられます。例えば、if 文、case 文、always プロシージャ、各種オペレータ等の使用法を論理合成ルールに従い正確に理解して使用する事により、目的とする RTL 設計を達成する事ができます。これに対して、検証分野における SystemVerilog の使用となると、事情が全く異なります。

検証分野においては、設計したデザインを検証するためのテスト環境を正確に、かつ効率的に表現し、しかも完全な検証を行わなければなりません。そのためには、SystemVerilog が備える機能を的確に選択して検証に適用する知識、技術、手法が必要になります。一方、SystemVerilog には検証機能、および検証に使用できる機能が多く備えられていますが、それらの機能を使用するための原理、原則、およびルールに関して SystemVerilog ではそれ程明確にされていません。例えば、組み合わせ回路やシーケンシャル回路からのレスポンスをサンプリングする最適なタイミングに関して、SystemVerilog として厳格なルールや推奨する方策が定められているわけではありません。確かに、LRM を精読すれば最適なタイミング候補として、checker インスタンス、program インスタンス、クロッキングブロック等が解決策になり得る事を突き止める事ができますが、その結論に到達するまでに果てしないと思える程の努力と時間が費やされます。したがって、万人が最適解に到達する事ができる訳ではありません。この点において、検証作業には難しさがあります。

近年では、検証作業の分業化・専門化が進みつつあり、検証技術者には従来よりも更に専門的な知識が要求されるようになっています。例えば、アサーションが一例として挙げられると思います。デザイン内に隠れた問題を解決するための的確なアサーションを準備するのは専門家のみが成し得る技であると言えます。あるいは、ランダムスティミュラスを巧みに生成して、設計時には想定していなかった現象を導き出す事により設計ミスを表面化する検証手法は、まさに専門家の仕事と言えます。

以上述べたように、一般的には、検証とは深遠な作業過程を意味しますが、所謂検証は至る所に存在します。記述したコードが正しく動作するか否かを確認する意味での検証は典型的な例です。しかも、この種の検証は誰もが遂行しなければならない作業です。本書の目的とする所は、このような広義の意味での検証に必要とされる SystemVerilog のプログラミング技術を解説する事です。本書では、多くの場合、解くべき問題を提起しそれに対する解決策を提示します。時には、複数の解決策が紹介されます。これに対して、読者自身が臨機応変に最適な使用法を選択すれば良いと思います。

本書は、検証分野で必要となるプログラミング技術を詳しく解説するために、データタイプに関する基本知識の応用から始まり、プロセス、プロセス間交信機能等の検証作業に不可欠な知識の解説に進み、次第に複雑な技術の解説へと移行します。その過程において、インターフェース、クラス、ランダムスティミュラスの生成などで遭遇する問題解決法と効果的な応用技術を解説します。最終的には、UVM を使用した検証環境の構築法、ファンクショナルカバレッジの検証への適用、パッケージの開発法、およびテストベンチ開発法へと進みます。

本書は、概要を含めて 12 章から構成され、データタイプの基本的な使用法から、並列処理の記述法と制御法、インターフェースの使用法、検証分野でのクラスの代表的な使用法、UVM, ファンクショナルカバレッジ等の幅広い範囲にわたり検証分野の話題をカバーしています。本書の各章の内容は易から難へと進むように構成されていますが、各章の内容は比較的独立しているので、必要に応じて学習する章の順序を変更しても構いません。ただし、本書を理解するためには SystemVerilog の基礎知識が必要です。本書は、SystemVerilog の入門書ではないので、初歩的な解説は含まれていません。

本書の殆どの章には、LRM に対応する章が存在しますが、本書独自の章もあります。以下、

それらの章の内容と意義を概説します。

第 9 章の UVM は、検証環境の構築に UVM を適用する際の良い参考書の役割を果たせるように構成されています。例えば、シーケンスを階層的に構築して実用的なシナリオ作成技術を実装しています。更に、テストケースの選択をコマンドラインから指定できるように設計してあるので、実践で必要となる知識の具体的実装例となります。検証環境を構成するそれぞれの検証コンポーネントを省略せずに解説しているので、UVM の学習と復習にも適しています。

第 11 章は、汎用的な機能の開発法を具体的なパッケージとして例示してあります。それぞれの機能は、単なる SystemVerilog 記述例の紹介ではなく、実用的な機能として実装されています。それぞれの実装コードからプログラミング技術を学ぶ事が可能であるだけでなく、それらの機能を活用して読者自身のツールへと進化させて行く方法もあると思います。

第 12 章は、テストベンチの開発法をまとめた章です。検証におけるタイミングの取り方の基本的な手法を簡潔にまとめてあるので、テストベンチ開発時の良い参考書になると思います。組み合わせ回路とシーケンシャル回路の両方に分けて検証法を記述しているので、この章を早い時期に読んでおくと効果的かも知れません。

本書には多くの例題が含まれ、しかもシミュレーション結果も添えられているので実行環境が十分に整っていない場所での学習にも適しています。ただし、見落とし易い機能記述もあるので、丁寧に学習を進めて下さい。

最後に、紙面の都合上、一部の記述は小さな書体で記述されています。

アートグラフィックス
篠塚一也

変更履歴

日付	Revision	変更点
2021.04.30	1.0	初版。

目次

本書で使用する略語一覧

略語	定義
ALU	Arithmetic Logic Unit
CRT	Constrained Random Test
DPI	Direct Programming Interface
DUT	Design Under Test、または、Device Under Test
EDA	Electronic Design Automation
FIFO	First In First Out
FSM	Finite State Machine
GCL	Generic Class Library
HDL	Hardware Description Language
LRM	Language Reference Manual、即ち、IEEE Std 1800-2017
RNG	Random Number Generator
RTL	Register Transfer Level
TLM	Transaction Level Modeling
UVM	Universal Verification Methodology

1　概要

本書の主題は検証分野で必要とされる SystemVerilog のプログラミング技術を解説する事ですが、検証作業ではハードウェア仕様、検証環境、およびそれらに伴う付帯条件等を考慮しながら適用すべき SystemVerilog 機能と実装方法を賢く決定しなければなりません。実際には、決定には幾つかの選択肢があり得ますが、最適な選択を導くためには検証機能の拡張性や再利用可能性等に関する考慮をすることにより選択の幅を狭めていく事が大切です。

検証作業は、与えられた DUT の動作がハードウェア仕様と一致するかを確認する事です。しかし、仕様だけからでは最適な検証条件を導けるとは限りません。以降では、DUT として 2 入力のマルチプレクサ `mux2` を例にとり、検証機能の実装選択が自明な作業ではない事を紹介します。ただし、自然言語による仕様定義では曖昧性と冗長性があるので、仕様を示す代わりに、DUT を SystemVerilog で記述する事にします。

1.1　検証例

以下に示すような 2 入力のマルチプレクサを検証する事を課題として与えられたと仮定します。

```
module mux2(input a,b,s,output logic q);

always @(a,b,s)
    if( s == 0 )
        q = a;
    else
        q = b;

endmodule
```

1.2　特別な条件がない場合の検証

特別な検証条件が与えられていなければ、以下に示すようなテストベンチでも有効であると考えられます。

例 1-1　特別な条件がない場合のテストベンチ例

以下に示すテストベンチには DUT のインスタンスおよび検証データが含まれています。DUT をテストする回数は便宜上 16 回に設定されていますが、特別な意味は持ちません。

以下の記述ではベクター `v` を使用して、DUT からのレスポンスを検証しています。`q==v[s]` であれば、DUT の結果は正しいと判断する事ができます。

```
module test;
logic     a, b, s, q;
logic [1:0]  v;
string    result;

mux2 DUT(.*);

initial begin
    repeat (16) begin
        #10;
        v = $random;
        s = $random;
        a = v[0];
        b = v[1];
    end
end
```

検証し易くするためにベクター v[1:0]を使用する

```
initial forever @(a,b,s)
    #0 $display("@%3t: %b %b %b %b %s",
           $time,a,b,s,q,q==v[s]?"pass":"fail");

initial $display("      a b s q");
endmodule
```

DUT からのレスポンスを検証する

ちなみに、このテストベンチを実行すると以下のような結果を得ます。

```
      a b s q
@ 10: 0 1 0 0 pass
@ 20: 0 1 0 0 pass
@ 30: 1 0 0 1 pass
@ 40: 1 0 1 0 pass
@ 50: 0 1 0 0 pass
@ 60: 0 1 0 0 pass
@ 70: 0 0 1 0 pass
@ 80: 0 0 1 0 pass
@ 90: 1 1 0 1 pass
@100: 1 1 0 1 pass
@110: 0 1 0 0 pass
@120: 1 1 1 1 pass
@130: 0 0 1 0 pass
@140: 1 0 0 1 pass
@150: 0 1 0 0 pass
@160: 0 0 0 0 pass
```

実行回数を増加すれば、より多くの組み合わせを試行する事ができ、このテストベンチは検証の役目を十分に果たすと考えられます。
■

確かに、上記のテストベンチでも目的を達成する事ができますが、もし仕様変更が発生して入出力が 4 ビットになった場合にはテストベンチにおける影響はどうなるでしょうか。すなわち、mux2 が以下のような仕様を持つ場合のテストベンチはどうなるでしょうか。

```
module mux2 #(NBITS=4) (input [NBITS-1:0] a,b,logic s,
    output logic [NBITS-1:0] out);

always @(a,b,s)
    if( s == 0 )
        out = a;
    else
        out = b;

endmodule
```

この場合、テストベンチの全面的な書き換えが必要になることが分かります。従って、例 1-1 で示したテストベンチは、仕様変更を考慮していなかった点に大きな問題があると言えます。

参考 1-1

SystemVerilog では、ビット数等は簡単にパラメータ化する事ができるので、検証機能を予めパラメータ化して置くことが大切になります。
□

1.3 仕様変更を考慮した検証

次に、仕様変更を念頭に置いてテストベンチを設計する事を考えます。テストベンチの改変を最小にとどめるために以下のような方策を採用します。

- トップモジュールとテストベンチを分離する。
- DUT の入出力ビット数の指定を可能にする。

トップモジュールとテストベンチを分離する事は、テストベンチの機能分離にもつながるので、検証環境の全容を把握し易くする効果があります。また、DUT の入出力ビット数の指定を可能にする事は、テストベンチを汎用化する事につながります。

例 1-2　仕様変更を考慮したテストベンチ

先ず、トップモジュールを以下のように定義します。DUT の入出力ビット数を変更してテストするためには、パラメータ SIZE を変更するだけで済みます。

```
module top;
parameter SIZE = 16;
logic [SIZE-1:0] a, b, q;
logic            s;

mux2 #(.NBITS(SIZE)) DUT(.*);
test #(.NBITS(SIZE)) TEST(.*);
endmodule
```

SIZE の指定を変えるだけで異なるテストを行える

一方、テストベンチを以下のように定義します。NBITS ビット変数に対するプリント処理があるため、書式設定が多少複雑になります。それ以外は、例 1-1 の記述と殆ど変わりがありません。

```
module test #(NBITS=4) (output logic [NBITS-1:0] a,b,logic s,
    input logic [NBITS-1:0] q);
import gcl_pkg::*;
parameter HEADER_WIDTH = (NBITS+3)/4;
logic [NBITS-1:0]   v[2];

initial begin
    repeat (16) begin
        #10;
        v[0] = $random;
        v[1] = $random;
        s = $random;
        a = v[0];
        b = v[1];
    end
end

initial forever @(a,b,s)
    #0 $display("@%3t: %h %h %h %h %s",
        $time,a,b,s,q,
        q==v[s]?"pass":"fail");

initial $display("      %s %s %s %s",
        make_header("a"), make_header("b"),
        "s", make_header("q"));

function string make_header(string item);
    make_header = gcl_sprint_left(item,HEADER_WIDTH);
endfunction
```

乱数を発生して DUT をドライブする

DUT からのレスポンスを検証する

結果を見易くするために書式を設定する

```
endmodule
```

表 1-1 はテストベンチで使用している重要なパラメータ、変数、およびファンクションの用途を示しています。NBITS の値設定に応じて自動的に書式が変更するように設計しています。このような概念をスケーラブルと呼びます。設計においても検証においてもスケーラブルなコードを記述する事は機能に拡張性を持たせるために重要な考え方です。

表 1-1　パラメータ、変数、およびファンクションの用途

定義項目	意味
parameter　HEADER_WIDTH = (SIZE+3)/4;	検証結果を 16 進数で表現するために必要な桁数を示しています。例えば、16 ビットの場合には 4 桁が必要となります。
logic [NBITS-1:0]　　v[2];	マルチプレクサの入力に対応する乱数発生用のアレイです。v[0]は a に、v[1]は b に対応するように使用しています。したがって、v[s]は DUT からの出力に一致する筈です。
function string make_header(string item);	結果を見易くするために、ビット数に合わせてプリント用のヘッダーを作り出しています。簡単に言えば、一時的に"%-Ns"という書式を作り出してプリント用のヘッダー文字列を構築します。ここで、N は 16 進数の桁数を示します。例えば、SIZE==16 の場合には N==4 なので、書式"%-4s"が作られ、"a"は左詰めの 4 文字から構成される文字列に変換されます。

このように定義すると、以下のような実行結果を得ます。

```
       a    b    s q
@ 10: 7cba a4e4 0 7cba pass
@ 20: 3e4c 1731 0 3e4c pass
@ 30: a301 c8a3 0 a301 pass
@ 40: ebe4 f1a2 0 ebe4 pass
@ 50: 2688 d5d7 0 2688 pass
@ 60: 0b17 4ad7 0 0b17 pass
@ 70: 9b47 7f20 0 9b47 pass
@ 80: a6e2 4017 1 4017 pass
@ 90: 2534 a061 1 a061 pass
@100: 3ac6 b8de 0 3ac6 pass
@110: c778 c8d0 1 c8d0 pass
@120: 2004 ea38 1 ea38 pass
@130: 8562 fc26 0 8562 pass
@140: 02a2 0320 0 02a2 pass
@150: 4a60 202d 1 202d pass
@160: 9ef6 4b13 0 9ef6 pass
```

なお、このテストベンチは SIZE==1 でも正しく動作するので、例 1-1 で紹介したテストベンチの役割も兼ねています。つまり、仕様変更を念頭に置いて、最初からこのようにテストベンチを設計しておけば、同じような作業を二度する事を回避できたと言えます。

■

さて、これで全てが完了したかと言えば、答えは"No"です。何故なら、乱数発生に使用している$random()システムファンクションは 32 ビットまでの整数しか取り扱う事ができないからです。つまり、例 1-2 で示したテストベンチは、32 ビット以内の入出力しか考慮していません。例えば、SIZE==35 として実行すると以下のような結果を得ます。乱数発生が 32 ビットまでなので、a、b、q の MSB は全て 0 になっています。

```
       a         b         s q
@ 10: 09a6f7cba 00d65a4e4 0 09a6f7cba pass
@ 20: 0cd3d3e4c 06fc51731 0 0cd3d3e4c pass
@ 30: 06175a301 07220c8a3 0 06175a301 pass
@ 40: 0d596ebe4 0a758f1a2 0 0d596ebe4 pass
@ 50: 0d4132688 03546d5d7 0 0d4132688 pass
@ 60: 06cde0b17 065aa4ad7 0 06cde0b17 pass
@ 70: 0575e9b47 004247f20 0 0575e9b47 pass
@ 80: 0814aa6e2 04c124017 1 04c124017 pass
@ 90: 0ccef2534 06873a061 1 06873a061 pass
@100: 0b6033ac6 08e07b8de 0 0b6033ac6 pass
@110: 0e093c778 0060cc8d0 1 0060cc8d0 pass
@120: 0870a2004 0a9bbea38 1 0a9bbea38 pass
@130: 0ad458562 0af37fc26 0 0ad458562 pass
@140: 0d0cf02a2 0c6b60320 0 0d0cf02a2 pass
@150: 0f41b4a60 02bb0202d 1 02bb0202d pass
@160: 0c3759ef6 05f174b13 0 0c3759ef6 pass
```

$random()を使用しているので、32 ビットの乱数しか発生されていない

結果を観察するとマルチプレクサの検証の役割を果たしていますが、テストベンチとしては正しいと言えません。もし DUT が 32 ビット以上の入出力を許す仕様であれば、このテストベンチでは不十分です。ビット数は、主として、プラットフォームに依存する傾向があるので、テストベンチはプラットフォームに独立な性質が求められます。次に、この課題を解決します。

1.4 プラットフォーム独立なテストベンチ

$random()システムファンクションの限界を克服するためには、std パッケージの randomize()ファンクションを使用する事ができます。このファンクションは、任意の変数をランダム変数として扱うことができる特長を持っています。次に、このファンクションの効果を紹介します。

例 1-3 プラットフォームに依存しないテストベンチの記述法

以下に示すように、例 1-2 における$random()システムファンクションを std::randomize()ファンクションで置き換えるだけで任意数のビットに対応する事ができるようになります。

```
module test #(NBITS=4) (output logic [NBITS-1:0] a,b,logic s,
    input logic [NBITS-1:0] q);
import gcl_pkg::*;
parameter HEADER_WIDTH = (NBITS+3)/4;
logic [NBITS-1:0]   v[2];

initial begin
    repeat (16) begin
        #10;
        assert( std::randomize(v) );
        s = $random;
        a = v[0];
        b = v[1];
    end
```

std::randomize()は v をランダム変数に変える能力がある

3 プロセス

3.1 実行順序依存性

SystemVerilog はハードウェアを記述するための並列処理言語で、initial や always プロシージャにより並列処理を表現する事ができます。それらのプロシージャは独立に実行するため、何らかの同期をとりながら処理を進めなければなりません。本書では、プロシージャが実行環境を与えられて実行するとき、プロセスと表現ずることにします。ただし、プロセスの一部の小さな実行単位を特定する場合には、スレッドとも表現する事にします。プロセスは、いつも活動しているとは限りません。通常は、プロセスにイベント待ち、またはディレー制御を含みます。

SystemVerilog では、プロセスがイベント待ち、またはディレー制御に遭遇して実行を中断すると、そのプロセスは実行権を放棄するため他のスケジューリングされているプロセスに実行権が移ります。例えば、以下のような簡単なシーケンシャル回路を仮定します。

```
module dut(input clk,d,output logic q,qbar);

assign qbar = ~q;       // p1

always @(posedge clk)   // p2
   q <= d;

endmodule
```

この記述には二つのプロセス（p1 と p2）が含まれていますが、何れのプロセスも実行待ちの状態にあります。p1 は、信号 q の変化を待っている状態にあり、p2 はクロック信号 clk の変化を待っている状態にあります。

もし(posedge clk)のイベントが起これば、p2 の待ち状態が解除されて q に d の値が設定されます。もし q の値に変化があれば、p1 の待ち状態が解除されて qbar の値も更新されます。一方、q に変化がなければ p1 の待ち状態は解除されないので qbar の値も変化しません。それぞれのプロセスが実行を再開しても、処理を終了すると再び待ち状態に入り同様の事を繰り返します。このように、プロセスが実行する時間は僅かであり、殆どの時間が実行待ちの状態に割かれています。

プロセス p2 の待ち状態の解除は外部から行われなければならないので、テストベンチを以下のように準備します。

```
module test(input clk,q,qbar,output logic d);
string judgement;

clocking cb @(posedge clk); endclocking

initial begin                   // p3
   for( int i = 0; i < 4; i++ ) begin
      d = i;
      @(negedge clk);
   end
   $finish;
end

initial begin                   // p4
   $display("     d q qbar judgement");
end

always @cb begin                // p5
   judgement = (d == q) && (q == !qbar) ? "pass": "fail";
```

```
    $display("@%2t: %b %b %b    %s",$time,d,q,qbar,judgement);
end

endmodule
```

テストベンチには三つのプロセス（p3、p4、と p5）が含まれていますが、どのプロセスが最初に実行するかは分かりません。p3 は信号 d に値を設定しますが、p2 の待ち状態を解除するわけではありません。p4 はメッセージをプリントするだけなので、p2 との関連はありません。そして、p5 はクロッキングブロックのイベント待ち状態になっています。何れにしても、テストベンチのどのプロセスも p2 の実行には影響を与えません。

次に、トップモジュールを以下のように準備します。トップモジュールは、先の二つのモジュールのインスタンスを作り接続します。そして、プロセス p6 においてクロックを生成します。したがって、p6 が p2 をドライブする事になります。

```
module top;
bit    clk;
logic  d, q, qbar;

dut DUT(.*);
test TEST(.*);

initial forever #10 clk = ~clk;   // p6

endmodule
```

以上紹介した六つのプロセスは、同時に実行を開始しますが、開始後まもなく待ち状態に入り活動を休止します。定期的に自発的に活動を再開するのは p6 だけです。さて、シミュレーション開始時のプロセス実行開始順序は実行環境に依存しますが、ここで紹介したプロセスは実行環境に依存せずに正しいシミュレーション結果を生成します。常に、実行環境に依存しないようにイベント制御を記述しなければなりません。

次に、実行環境に依存する記述例を考えてみます。例として、以下のような記述を仮定します。この例では、consumer と producer の二つのプロセスが生成されて実行しますが、どちらが先に実行するかは実行環境に依存します。

```
module test;
event  get_item;

initial begin
    fork
        consumer;      ┐   どちらのプロセスが先に実行
        producer;      ┘   するか分からない
    join
end

task consumer();
    forever begin
        ->get_item;
        ...
    end
endtask

task producer();
    forever begin
        @get_item;
        ...
```

4　プロセス間交信機能

SystemVerilog で並列処理を制御するための機能は、表 4-1 にまとめられています。これらの機能を使用して並列処理を効果的に実現する事ができます。

表 4-1　プロセス間同期と通信機能

プロセス間同期機能	仕様
semaphore	セマフォは、共有資源にアクセスするための排他制御機能を提供します。セマフォにはキーの数 K を割り当て、K 個のキーを使用して排他制御を行います。空いているキーが存在する限り、プロセスは共有資源にアクセスする事ができます。使用可能なキーが存在しない場合、キーを取得するプロセスはキーが使用可能になるまで待たされます。
mailbox	メールボックスは、並列に実行しているプロセス間でメッセージを交換する仕組みです。あるプロセスがメールボックスにメッセージを書き込むと、別のプロセスがメールボックスからメッセージを取得する事ができます。しかし、メッセージがまだ届いていない場合には、取得側は待つか、別の機会に再度メッセージを取得するようにしなければなりません。メールボックスは、有限または無限のサイズを持つように定義する事ができます。有限の大きさを持つメールボックスは、メッセージの量が許容量に達すると、メッセージを書き込むプロセスはメールボックスに空きができるまで待たなければなりません。一方、無限のサイズを持つメールボックスの場合には、メッセージは常に書き込み可能になります。メッセージとしては、数値、文字列、およびクラスオブジェクト等を指定する事ができます。メールボックスには、メッセージを書きこむプロセスとメッセージを取得するプロセスが独立に処理を進める事ができる利点と、書き込まれたメッセージが失われることなく取得されるという利点があります。
event	イベントは、データやメッセージを使用せずにプロセス間の同期をとる機能です。プロセスは、@identifier、または wait()文でイベントが発生するのを待ち、別のプロセスが->identifier で待ち状態を解除します。通信対象のプロセスを意識せずにプロセス制御をする事ができる利点があります。

セマフォとメールボックスは、std パッケージに定義されているビルトインのデータタイプですが、クラスなのでユーザは自由に拡張する事ができます。一方、event データタイプは、Verilog にも存在する機能ですが、SystemVerilog では大きな機能拡張が実現されています。このデータタイプに関する機能は、前章までに多くの例を紹介したので、本章では簡単な解説に留めます。

4.1　semaphore

4.1.1　概要

セマフォは SystemVerilog のクラスとして実現されています。クラスハンドルを定義して、ハンドルに new コンストラクタを使用してオブジェクトを割り当てる手順を取ります。例えば、以下のようにしてセマフォを宣言して使用可能な状態にします。

```
semaphore mutex;
mutex = new(5);
```

この場合には、5 個のキーを持つセマフォが定義されます。もし一つのプロセスにつき 1 個の

キーを使用するとすれば、同時に 5 個までのプロセスが共有資源のアクセスできることになります。しかし、6 個目のプロセスには共有資源へのアクセス権は与えられません。

セマフォには表 4-2 に示すようなメソッドが定義されています。put()と get()メソッドはブロッキングメソッドですが、try_get()メソッドはノンブロッキングメソッドです。

表 4-2　セマフォのメソッド

メソッド	意味
function new(int keyCount = 0);	指定した数のキーを持つセマフォを作成します。バケットは指定されたキーの数で初期化されます。
function void put(int keyCount = 1);	指定した数のキーをセマフォに戻します。戻すキー数の標準値は 1 です。
task get(int keyCount = 1);	指定した数のキーを取得します。指定数のキーが存在すれば呼び出しているプロセスの実行は継続します。もし、指定数のキーが存在しない場合には、呼び出しているプロセスの実行がブロックされ、キーの取得が可能になるまで待ち状態になります。
function int try_get(int keyCount = 1);	指定した数のキーを取得します。ただし、呼び出しているプロセスをブロックしません。もし、指定数のキーが存在すれば正の整数を戻します。もし、指定数のキーが存在しない場合には 0 を戻します。

参考 4-1

もし、以下のようにセマフォを確保すると、キー数が 0 のセマフォ lock が作られます。

```
semaphore lock;
lock = new;
```

lock にはキーが一つも存在しないため、以下の lock.get(1)は永久に待ち続ける可能性があります。

```
lock.get(1);
$display("@%0t: got a key",$time);
```

一見、この記述は意味が無いように思えますが、そうとも言えません。何故なら、他のプロセスが以下のようにキーを作り出すことができるからです。

```
lock.put(1);
```

この命令が実行されると、セマフォ lock には一つのキーが増加されるため、上記の lock.get(1)が解除されます。
□

4.1.2　セマフォによる実行順序依存性の回避

3.6 節では、イベントや fork ブロックにより実行順序を決定する方法を紹介しましたが、ここではセマフォを使用して並列プロセスの実行順序を決定する方法を紹介します。セマフォの効果を明確にするために、例 3-6 を以下のように書き換えます。

6　インターフェース

インターフェースは、設計と検証の何れの分野でも使用される重要な機能です。設計分野ではインターフェースは、主として、モジュール間の接続に柔軟性を持たせる目的とポートの維持を容易にする目的のために modport が使用されます。検証分野ではインターフェースの殆どの機能が使用されますが、とりわけ以下のような機能が重要な役割を果たします。

- virtual インターフェース
- クロッキングブロック
- modport

本章では、インターフェースを検証分野で使用する重要な手段として近年の検証技術と関連をさせながら特徴的な使用法を解説します。

6.1　モジュールによる検証環境

先ず、インターフェースがデザインに使用される環境を考える事から始めます。簡単のために、以下のようなインターフェースが準備されていると仮定します。

```
interface simple_if(input logic clk);
...
clocking cb @(posedge clk); endclocking
modport DUT (...);
modport TEST (...);
endinterface
```

デザインは、インターフェースを以下のように使用します。

```
module device(simple_if.DUT mp);
...
endmodule
```

このデザインを検証するためのテストベンチは以下のようになります。以下の記述例では program を使用していますが、代わりに module を使用する事もできます。

```
program test(simple_if.TEST mp);
...
endprogram
```

次に、テストベンチとデザインと接続するためのトップモジュールを以下のように準備します。

```
module top;
bit    clk;

simple_if SIF(.*);
test TEST(SIF);
device DUT(SIF);
...

endmodule
```

これらのインスタンスはコンパイル時に決定していなければならないので、検証環境の柔軟性に欠ける

simple_if のインスタンス SIF が示すように、インターフェースは、モジュールと同様にインスタンスを作らなければ機能しません。そして、インターフェースのインスタンスの配置はスタティックに行われコンパイル時に決定していなければなりません。同様に、ここで示すテストベンチのインスタンスもスタティックに配置されています。換言すれば、トップモジュールの構成はコンパイル時には決定されており、シミュレーション実行時に変更する事はできません。

上記に示したようなスタティックな検証環境では検証作業の生産性が低いのは明らかです。例えば、N 個のテストケースがあれば、N 個のテストベンチに対して N 個のトップモジュールが必要になります（図 6-1）。この場合、テストを実行するためには少なくとも N 回のコンパイル作業が必要になります。テストベンチをスタティックに配置する限りこの状況を回避する事はできません。モジュールや program のインスタンスを使用した検証環境は、柔軟性に欠けている事が分かります。

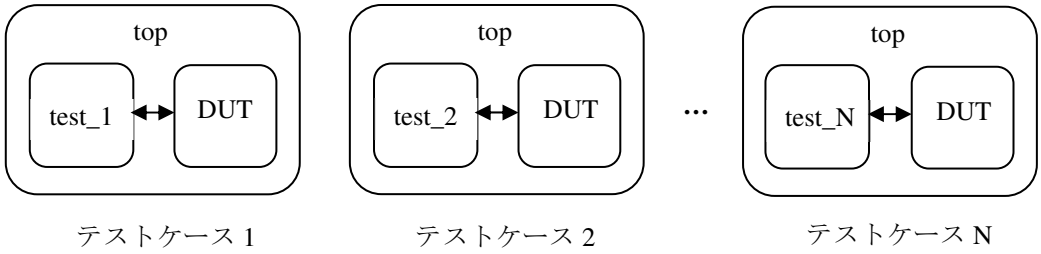

図 6-1　モジュールによる検証環境構成例

6.2　近年の検証手法

近年の検証手法は、モジュールや program を使用する代わりにクラスを使用して検証環境を構築します。クラスの場合にはインスタンスをダイナミックに作る事ができるので、N 個のテストケースに対して唯一つのトップモジュールで十分です。したがって、1 回のコンパイルで N 個のテストを行えるため、検証作業は効率良く進められます。クラスを使用すると、トップモジュールの構成は以下のようになります。

```
module top;
bit    clk;

simple_if SIF(.*);          } デザインに必要な情報だけをスタティックに配置する
device DUT(SIF);            }

initial begin
    setup_config();         } SystemVerilog のクラスを使用して検証環境を実行時に決定する
    run_test();             }
end
...

endmodule
```

ここで、setup_config()はコンフィギュレーションを設定してテストケースの準備をする役割を果たし、run_test()はテストケースをコマンドラインから取得してシミュレーションを実行する役割を果たします。コンフィギュレーションには、virtual インターフェース情報が含まれます。

このアプローチによれば、インスタンスとして必要なのはインターフェースのインスタンスと DUT のインスタンスだけです。したがって、テストケースに依存する情報はスタティックに使用されていないため、実行時にテストケース情報を決定する事ができます。通常は、テストケースや検証環境を決定するパラメータ値をコマンドラインから設定するようにして、使用法に柔軟性を持たせます（図 6-2）。図は、テストケースとして test_2 が選択されている状態を示し、vif は virtual インターフェースを示しています。クラスによる検証環境では、virtual インターフェースを介して DUT のドライブ、DUT からのレスポンスのサンプリングを行います。その際、クロッキングブロックは非常に有効な手段となります。

9 UVM*

本章では検証環境に UVM を適用する例を紹介します。既存の検証環境構築法との対比を明確にするため、6.5 節で紹介した構築例に従い UVM を使用して再構築します。なお、以下の解説では UVM に関する基礎知識を仮定しています。

9.1 UVM による検証環境

図 9-1 に示すような検証環境を構築します。また、表 9-1 は検証環境の主な構成要素の機能を示しています。

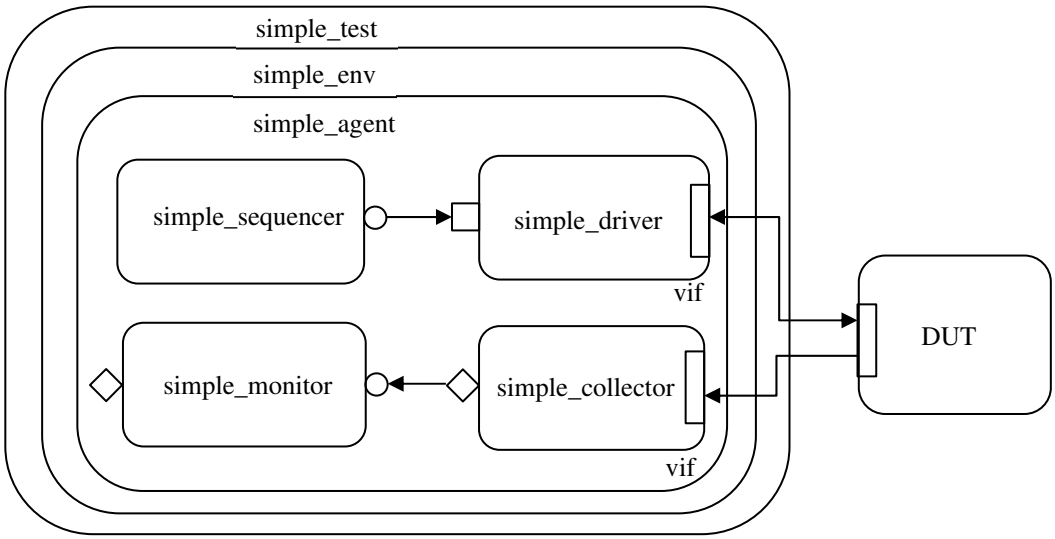

図 9-1　UVM による検証環境構築例

表 9-1　検証環境を構成する主な要素

構成要素	機能
simple_if	6.5.1 項で定義されたインターフェースクラスをそのまま使用します。
simple_item_t	トランザクションを示すクラスです。
simple_sequence_reset_t simple_sequence_load_t simple_sequence_up_t simple_sequence_down_t	トランザクションは RESET、LOAD、UP、DOWN 等のコマンド機能を備えているので、それらのコマンドに対応するシーケンスを定義しておきます。
simple_sequence_base_t	テストケースのベースクラスです。
simple_sequence1_t	テストケース 1 のシナリオを生成するクラスです。
simple_sequence2_t	テストケース 2 のシナリオを生成するクラスです。
simple_driver_t	DUT をドライブするドライバーのクラスです。
simple_sequencer_t	ドライバーの要求によりトランザクションを生成するクラスです。このクラスの run_phase()には、default_sequence としてテストケースのシナリオが割り当てられます。
simple_collector_t	DUT からのレスポンスをサンプリングするコレクターのクラスです。
simple_monitor_t	コレクターから受信したトランザクションの処理をします。この例では、トランザクションをプリントするだけの簡単な処理内容になっています。
simple_agent	ドライバー、シーケンサー、コレクター、モニターから構成されるエージェントのクラスです。
simple_env	エージェントから構成されるトップレベルのエンバイロンメント

```
task simple_sequence1_t::body();
    `uvm_do(seq_reset)
    `uvm_do(seq_up)
    `uvm_do(seq_up)
    `uvm_do(seq_up)
    `uvm_do(seq_load)
    `uvm_do(seq_up)
    `uvm_do(seq_down)
endtask
```

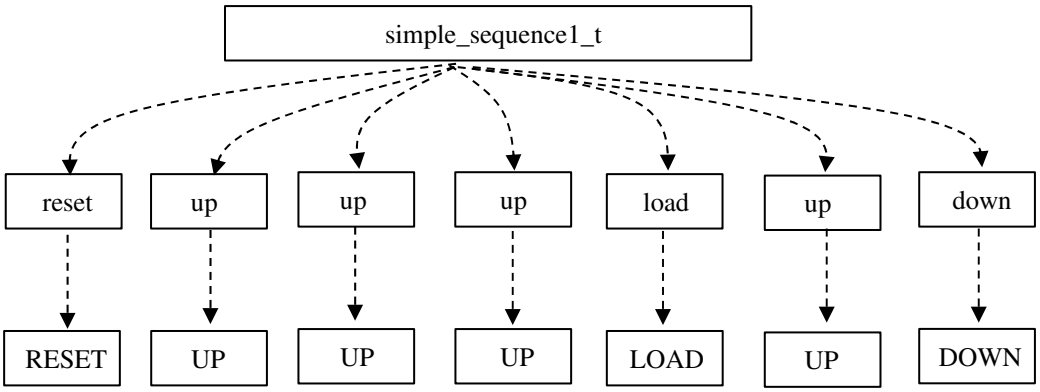

図 9-2　テストケース 1 のシーケンス階層

9.2.8　simple_sequence2_t

このクラスはテストケース 2 の内容を定義します（図 9-3）。body()タスクが主な内容となります。

```
class simple_sequence2_t extends simple_sequence_base_t;

`uvm_object_utils(simple_sequence2_t)

function new(string name="simple_sequence2_t");
   super.new(name);
endfunction

extern virtual task body();
endclass
```

body()タスクは以下のようになります。以下の記述にあるように、テストケース 2 は LOAD コマンドから開始します。

```
task simple_sequence2_t::body();
    `uvm_do(seq_load)
    `uvm_do(seq_up)
    `uvm_do(seq_down)
    `uvm_do(seq_reset)
    `uvm_do(seq_up)
    `uvm_do(seq_up)
    `uvm_do(seq_down)
endtask
```

11　パッケージの開発法

SystemVerilog によるプログラミング例として、本章では汎用的な機能を持つタスクとファンクションの記述例を紹介します。ただし、それらの機能は単なる記述例ではなく実践において有効に活用できるように設計します。また、機能を記述する際、特殊な技巧を用いずに標準的な SystemVerilog 機能で記述するように心がけているので、開発環境に依存しない汎用的な機能を開発する参考例になると思います。

また、実践では検証作業で必要となる多くの機能をパッケージとして定義する機会が多く出て来ると思われるので、本章で開発する機能群をパッケージとしてまとめ、パッケージ開発時の参考例となるように構築してあります。参照の便宜を図るため、以降で紹介する機能を GCL パッケージとして定義してあります。なお、本章で紹介する記述例は機能において限定的な側面を持つ可能性があるため、読者自身の目的に適合するように機能を拡張・変更する事を勧めます。むしろ、そのような過程を経て SystemVerilog によるプログラミング技術を向上する事ができると思えます。

11.1　GCL パッケージの概要

GCL パッケージには表 11-1 に示す機能が定義されています。

表 11-1　GCL パッケージに定義されている機能

名称	種別	機能
gcl_clock_gen	ファンクション	クロックを生成しタイムアウトを管理する機能を持ちます。タイムアウトが発生するとシミュレーションを終了させます。ファンクションなので、呼び出し側に即座に戻りますが、クロック生成プロセスは実行を続けます。
gcl_dlink_t	クラス	双方向ポインターを持つデータ構造です。このクラスのオブジェクトは双方向リンクリストを構成します。
gcl_dlist_t	クラス	双方向リンクリストを管理するクラスです。
gcl_make_format	ファンクション	指定した幅を持つ書式を作ります。
gcl_max_len	ファンクション	文字列のアレイに対して、文字列の最大長を戻します。
gcl_process_manager_t	クラス	任意個のプロセスを作る出す機能を持つクラスです。
gcl_process_t	クラス	gcl_process_manager_t により作られたプロセスを表現するクラスです。
gcl_random_string	ファンクション	文字列をランダムに発生します。
gcl_reverse_string	ファンクション	与えられた文字列を逆順にした文字列を戻します。元の文字列には影響はありません。
gcl_sprint_center	ファンクション	文字列を中央に揃えます。
gcl_sprint_left	ファンクション	左詰めにした文字列を戻します。
gcl_sprint_right	ファンクション	右詰めにした文字列を戻します。
gcl_sprint_string	ファンクション	文字列をカラムに揃えた文字列を戻します。

11.2　GCL パッケージ使用法

GCL パッケージを使用するためには、以下の手順が必要です。

- `gcl_pkg.sv` ファイルをインクルードする。
- パッケージを使用するスコープ内で、`gcl_pkg` をインポートする。

例えば、以下のように使用します。

```
`include "gcl_pkg.sv"

module test;
import gcl_pkg::*;
...
endmodule
```

11.3 GCL パッケージの構成

GCL パッケージは gcl_pkg.sv ファイルに定義されていますが、このファイルは以下のような内容で構成されています。それぞれのファイルの意味を表 11-2 に示します。

```
`ifndef   GCL_PKG_H
`define   GCL_PKG_H

package gcl_pkg;

// declaration and common variables
`include "gcl_definitions.sv"
`include "gcl_global.sv"

// generic
`include "gcl_string.sv"
`include "gcl_sprint.sv"

// utility
`include "gcl_random_string.sv"
`include "gcl_clock_gen.sv"

// list
`include "gcl_dlink.sv"
`include "gcl_dlist.sv"

// process
`include "gcl_process.sv"
`include "gcl_process_manager.sv"

endpackage

`endif
```

表 11-2　GCL パッケージを構成するファイル

ファイル	意味
gcl_definitions.sv	このファイルは、GCL パッケージに定義されている機能が使用するデータタイプの宣言を含みます。
gcl_global.sv	GCL パッケージで共有する定数を定義するためのファイルです。現在では GCL パッケージの情報だけを含みます。
gcl_string.sv	文字列に関する汎用機能を含むファイルです。
gcl_sprint.sv	汎用的な書式機能を含むファイルです。
gcl_random_string.sv	ランダム文字列を発生する機能を含むファイルです。
gcl_clock_gen.sv	クロック生成機能を含むファイルです。
gcl_dlink.sv	双方向リストを構成するリンクアイテムのデータ構造を含むファイルです。
gcl_dlist.sv	双方向リストを管理するためのファイルです。
gcl_process.sv	gcl_process_manager_t により作られるプロセスを表現するクラ

12　テストベンチ

これまでの章においてテストベンチを何度となく紹介しましたが、この章ではテストベンチ記述法を総括する意味でまとめておきます。

デザインを記述すると検証をしなければなりません。システム全体、または他のブロックとの組み合わせテストをする前に、設計者は書き終えたデザイン自身のテストを行わなければなりません。以下では、このレベルのテストを単体テストと呼ぶ事にします。単体テストはデザインを記述した設計者自身が行うものであるため、設計者には SystemVerilog が備える検証機能に関する知識が要求されます。本章では、単体テストに必要な基礎知識を解説します。

12.1　概要

デザインは組み合わせ回路、シーケンシャル回路、FSM 等の種類に分かれますが、それぞれの検証には特殊性が含まれます。例えば、組み合わせ回路の場合には、入力の変化に応じて出力の計算が行われ、クロックと非同期に出力結果が求まります。その際、出力をどのようなタイミングで検証するかが重要な問題となります。仮に、検証の対象である組み合わせ回路がハーフアダーで、その出力を co と sum とします。もし、以下のように検証のタイミングを選ぶと、正しい検証はできません。

```
always @(co,sum) begin
...
end
```

なぜなら、このイベント制御では、co、または sum に変化がなければ always プロシージャの内部が実行されないからです。ハーフアダーの入力 a と b に変化があるにも関わらず検証が行われない事になります。

表 12-1 は、ハーフアダーの真理値表を示しています。表の中でハイライトされた二行に着目して下さい。co と sum の値には変化がありませんが、ハーフアダーの入力 a と b の値には変化があります。従って、上記の always プロシージャはハーフアダーの入力 a と b の変化をとらえていない事が分かります。換言すれば、上記の検証するタイミングは正しくないと言えます。

表 12-1　ハーフアダーの真理値表

a	b	co	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

次に、以下のように変更するとします。

```
always @(a,b) begin
...
end
```

しかし、この検証法は正しくありません。すなわち、以下のような問題点があります。

- ハーフアダーの計算が完了する前に、always プロシージャが実行するため、古い計算結果を参照する可能性がある。
- always プロシージャ内では、a と b 以外の変数を参照する事ができない。例えば、co、または sum を always プロシージャ内で参照すると、コンパイラーが警告を発行してしまう。つまり、always プロシージャのセンシティビティリストは完全でなければならない。

したがって、以下のように書き換える事が必要になります。

```
initial forever @(a,b) #0 begin
...
end
```

一方、$monitor タスクを使用すると問題は解決するように思えますが、完全な解決策にはなり得ません。例えば、以下のように検証をすると仮定します。

```
initial $monitor("@%2t: %b  %b   %0d",$time,a,b,{co,sum});
```

確かに、入力の変化に対応して結果がプリントされますが、$monitor タスクは Postponed 領域で実行するため、組み合わせ回路の計算タイミングを正しく判定する事ができません。例えば、ハーフアダーを以下のように記述しても$monitor タスクは正しい結果をプリントします。

```
module half_adder(input a,b,output logic co,sum);

always @(a,b)
    {co,sum } <= a+b;

endmodule
```

ノンブロッキング代入文を使用しているので、
正しい組み合わせ回路の記述ではない

これに対して、ディレー#0 を使用した initial プロシージャによる検証方法は、この half_adder 記述が間違いである事を判定する能力を持っています。

以上は組み合わせ回路のテストに関しての注意ですが、シーケンシャル回路の場合にも同様な課題があります。すなわち、シーケンシャル回路の出力をサンプリングするタイミングは競合状態を避けるようにしなければなりません。

以降では、これらの課題を考慮したテストベンチ構築法を紹介します。ただし、SystemVerilog の program ブロックを使用しない手法を採用します。program ブロックは特異性を持つため、使用が制限されている検証環境が多く存在します。例えば、UVM では program ブロックを使用する事ができません。

12.2 組み合わせ回路

簡単な組み合わせ回路を例にとりテストベンチ構築法を解説します。

12.2.1 デザイン例

以下のようなフルアダーを検証すると仮定します。

```
module full_adder(input a,b,ci,output logic co,sum);

always @(a,b,ci)
    {co,sum} = a+b+ci;

endmodule
```

12.2.2 検証法

12.2.2.1 モジュールによる検証

以下に示すように、モジュールの検証法は標準的な構造を持ちます。

- 検証すべきデザインのインスタンスを作成する。
- デザインが使用する信号名を宣言する。
- デザインの入力を準備するプロシージャを準備する。