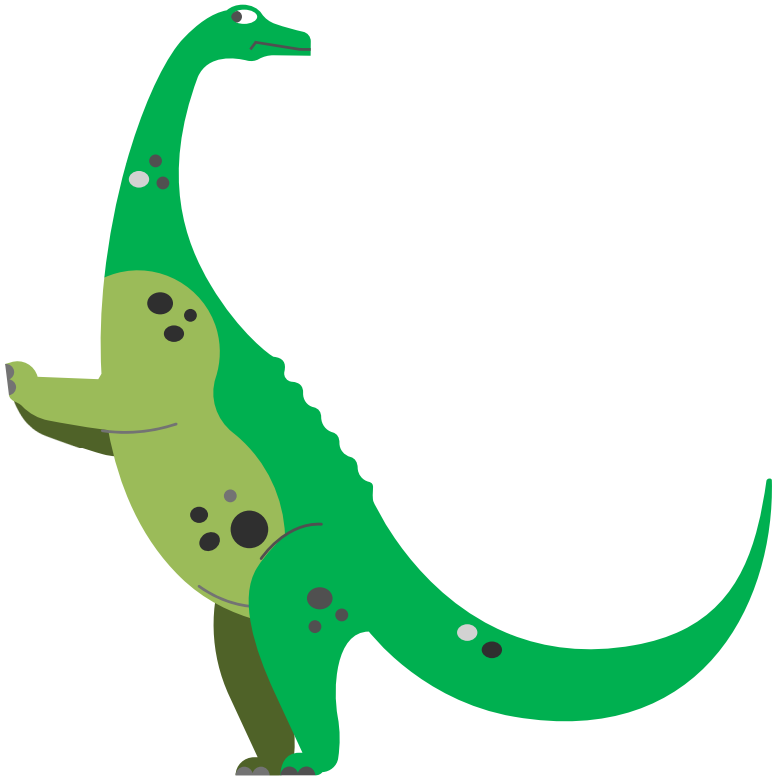
Artgraphics

SystemVerilog シミュレーションの論理

ハードウェア記述言語の動作解析のための基礎知識

Document Identification Number: ARTG-TD-002-2024
Document Revision: 2.0, 2024.02.25
アートグラフィックス
篠塚一也



SystemVerilog シミュレーションの論理
ハードウェア記述言語の動作解析のための基礎知識

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

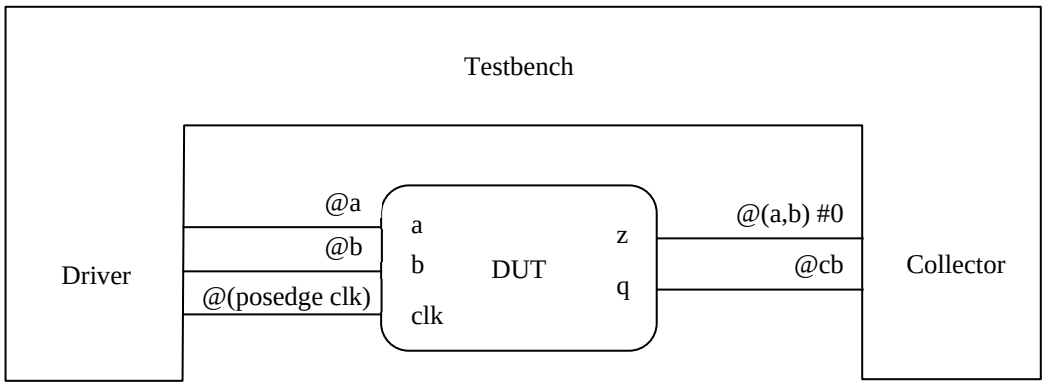
Simulation Semantics for SystemVerilog

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

はじめに

本書は、SystemVerilog シミュレーション時に発生するスケジューリング、実行順序、タイミング等に関する規約を詳しく解説した技術資料です。対象となる読者は、初級から中級レベルの技術者です。本書を理解するためには SystemVerilog の基礎知識が必要ですが、高度な知識を持つ必要はありません。例えば、ランダムスティミュラスの生成、ファンクショナルカバレッジやアサーション等の知識を持たなくても本書を理解する事ができます。

デザインを検証するためには、SystemVerilog のシミュレーション論理を正しく理解しなければなりません。例えば、DUT をテストする際、DUT が回路の入力信号に関して入力待ちの状態になってから、ドライバーは DUT をドライブしなければなりません。さもなければ、DUT は信号値の変化を見逃してしまいます。特に、この問題は`$time==0` における非同期リセット信号に関して発生し易い現象です。同様に、DUT のレスポンスを監視するコレクターの状態が整ってから、ドライバーは DUT をドライブしなければなりません。下図は、その状況を示しています。DUT にしてもコレクターにしても reactive システムなので、それらのイベント待ちが有効になってから、ドライバーが DUT をドライブしなければなりません。



clocking cb @(posedge clk); endclocking

したがい、以下に示すような記述に対しての効果の相違を理解しておかなければなりません。

```
always @(a,b) ...
always @(posedge clk) ...
forever @(a,b) #0 ...
forever @cb ...
```

組み合わせ回路では、Active 領域で `a` または `b` の信号値の変化が起こると`@(a,b)` のイベント待ちが解除され後続の命令が実行を開始します。実行後に LHS にある信号値の変化が引き起こされるのは矢張り Active 領域なので、その後の Inactive 領域で LHS の信号値の変化を検証するのが適切なタイミングです。まとめると、以下のようになります。

組み合わせ回路	検証

`always @(a,b)`
`  z = a&b;`
	`initial forever @(a,b) #0 begin`
	`  if( z != a&b )`
	`  ...`
	`end`

組み合わせ回路の出力は Active 領域で決定する。	Inactive 領域では組み合わせ回路からの出力が安定しているので検証可能です。

同様に、シーケンシャル回路のクロック信号や非同期リセット信号は Active 領域で変化します。したがって、(`posedge clk`)や(`posedge reset`)等のイベントが Active 領域で発生します。しかし、イベント待ちに続く命令の中にはノンブロッキング代入文が含まれているので、その LHS の信号値の変化は NBA 領域で起こります。そのため、組み合わせ回路とは異

なる検証のタイミングが必要になります。

シーケンシャル回路は、同じ$time の間に Active 領域と NBA 領域の実行を繰り返すため NBA 領域のどの時点で LHS が変化するかを突き止める事は不可能です。したがって、NBA 領域以降で LHS の検証をしなければなりません。最も適切なタイミングは Observed 領域であり、そのための機能が SystemVerilog により準備されています。その機能は、所謂、クロッキングブロックです。以下の表は、以上の説明のまとめです。

シーケンシャル回路	検証
`always @(posedge clk)` `  q <= d;`	`clocking cb @(posedge clk); endclocking` `initial forever` **`@cb`** `verify();`
シーケンシャル回路の出力は NBA 領域で決定する。	`@cb` は Observed 領域で解除されます。Observed 領域ではシーケンシャル回路からの出力が安定しているので検証可能です。

SystemVerilog は厳密なシミュレーション規約で構成されています。その規約は、以下のような簡単な記述にも適用されます。この記述において、変数 a の値がどのように変化するかを論理的、かつ明解に説明し、a の値の変化を計測する手順を正しく記述できる人は本書を読む必要はありません。しかし、説明に確信を持てない方、あるいは、この課題に技術な興味を持つ方は、本書を読むべきです。

```
fork
   a = 1;
   #0 a = 2;
   a <= 3;
join
```

機能が広範囲にわたる事を別として、SystemVerilog が難しいのはシミュレーションのセマンティックスが厳密に定義されている事と言えます。例えば、Verilog と異なり、SystemVerilog では、ネットや変数の宣言時に定義した初期値の設定は、シミュレーション開始直前に行われる事が挙げられます。更に、SystemVerilog では、シミュレーションをするタイミングを 17 個のスケジューリング領域に分割して管理しているので、それぞれの SystemVerilog 機能が動作するスケジューリング領域を正しく理解しておく必要があります。SystemVerilog の学習過程の始めのうちは、それらの分割が複雑に見えますが、スケジューリングの意味を理解するにつれて合理的な分割であると納得するように変化していく筈です。SystemVerilog の基本原則は、Verilog 時代に存在したシミュレーションセマンティックスの曖昧性を排除して、シミュレーションのスケジューリングを厳密に定義し直し、結果が実行環境や使用するツールに依存しないようにする事です。つまり、誰でもが疑いなく同じ解釈をできるように規約を厳密に定義しています。

本書は、どのような SystemVerilog 機能を使用するのが適切であるかを解説するのではなく、規約にしたがい機能をどのように使用するかに重点を置いています。したがって、機能に関する選択肢の解説ではなく、適切と考えられる機能を選び、その機能をどのように使用すると最適な処理を実現する事ができるかを議論して結論を導くように解説を進めています。

本書が対象とする読者は、SystemVerilog を学習し始めて間もない技術者です。ただし、読者は SystemVerilog に関する基本的な知識を持っていると仮定しています。例えば、データタイプ、fork-join、プロセス間通信機能（semaphore、mailbox、event 等）、クラス、インターフェース等の用語に馴染んでいる事が必要です。それらに対して完全な知識をもつ必要はありませんが、概要的な知識があると本書の内容を十分に理解できます。

SystemVerilog は、Verilog と異なり厳密な規約によりシミュレーション機能を実行して行きます。それらの規約は、LRM の何処かに明記されているのですが、初心者は規約の重要性を見逃してしまう場合が多く見られます。本書の目的は、そのような状況が発生しうる機能を重

点的に解説して初心者が間違いを未然に防げるようにする事です。例えば、実行環境に依存しない記述を身に付けるためには、基本的に備えるべき知識があります。本書の目的は、そのような知識・技術を論理的、かつ明解に解説します。

しかし、単に解説を読むだけでは十分に内容を理解する事ができません。読者自身で手法を体験する事が最も重要であり、かつ完全な方策です。そのために、本書では各章の終わりに練習問題を準備しました。問題を提示している目的は、本文内容を補足する意味と読者の理解を確認する意味があります。全体的には、SystemVerilog のシミュレーション規約に関する基本的な知識を尋ねていますが、必ずしも容易ではない問題も含まれています。全ての問題に対して詳しい解答が用意されているので、できるだけ多くの問題を解くようにして下さい。

洋書を含めて、本書は市販されている書物には書かれていない情報を多く含んでいます。これらの内容は、特に、SystemVerilog 初心者におすすめします。本書を読めば、SystemVerilog に関する知識が飛躍的に増大します。なお、初心者にはやや難しいと思われる内容を持つ節や問題には＊印が付いているので、それらの節や問題は丹念にお読み下さい。

アートグラフィックス
篠塚一也

変更履歴

日付	Revision	変更点
2021.07.10	1.0	初版。
2021.12.15	1.1	①　計算精度の仕組みの章を追加。 ②　補足の章を追加。
2024.02.25	2.0	①　全体的に章の再構成を実施。 ②　「式の計算精度」の章を削除。 ③　練習問題を追加。

目次

1 SystemVerilog によるシミュレーション

本章では SystemVerilog シミュレーション全体に関わる技術的な話題を概説して、本書の目的を明確にします。難易度が高い内容もあるので、記述に疑問を持つ場合には、自身で例を作り実験をしてみる事をすすめます。

1.1 テストベンチ

initial や always で記述されたプロシージャは、シミュレータにより標準プロセスとして起動されます。しかも、それらのプロセスの実行順序は SystemVerilog 言語では規定されていないため、どのプロシージャが先に実行するか予め決定する事はできません。したがって、プロシージャがどのような順序で実行を開始しても正しい実行結果を得るためには、プロセス間の同期を取る事により一定の実行順序を実現するか、あるいは SystemVerilog のシミュレーション論理に従い実行順序を理論的に確立しなければなりません。先ず、簡単な例を使用してプロセスの実行順序がもたらす問題を観察してみます。

例 1-1　プロセスの実行順序が実行環境に依存する例

ここでは、組み合わせ回路を検証する例を紹介しますが、どのような組み合わせ回路にも発生し得る現象であるので、簡単な組み合わせ回路を使用して検証例を構築します。ここでは、以下に示すような 1 ビットハーフアダーを検証します。a または b の値が変化すれば加算の演算が行われますが、変化がなければ演算は行われません。

```
module ha_bh(input a,b,output logic co,sum);
always @(a,b)
    {co,sum} = a+b;
endmodule
```

テストベンチを以下のように準備します。

```
module test;
logic  a, b, co, sum;

ha_bh DUT(.*);

initial begin: p1
    for( int i = 0; i < 4; i++ ) begin
        {a,b} = i;
        #10;
    end
end

initial begin: p2
    $display("time a b {co,sum}");
    forever @(a,b) #0
        $display("@%2t: %b %b    %0d",$time,a,b,{co,sum});
end

endmodule
```

最初の initial プロシージャ（p1）で DUT をドライブし、二番目の initial プロシージャ（p2）で検証のために DUT からの出力をサンプリングしています。実行すると、以下のような結果を得ます。

```
time a b {co,sum}
@10: 0 1    1
@20: 1 0    1
@30: 1 1    2
```

結果を観察すると、計算結果は正しいのですが$time==0 の結果がプリントされていない事に気が付きます。プロシージャ p1 で$time==0 において {a,b}に 0 をセットしているにも関わらず、$time==0 における結果がプリントされていません。その理由を以下のように説明できます。

プロシージャ p1 がプロシージャ p2 よりも先に実行を開始しているため、p2 におけるイベント待ち @(a,b) が有効になる前に、{a,b} に 0 がセットされています。したがって、$time==0 における {a,b} のイベント発生が p2 で見落とされています。

この問題を解決するためには、プロセス間の同期を取るか、あるいは p1 の実行開始を p2 の後にしなければなりません。因みに、p1 と p2 の配置順序を以下のように変更してみます。

```
initial begin: p2
    $display("time a b {co,sum}");
    forever @(a,b) #0
        $display("@%2t: %b %b    %0d",$time,a,b,{co,sum});
end

initial begin: p1
    for( int i = 0; i < 4; i++ ) begin
        {a,b} = i;
        #10;
    end
end
```

こうすると、以下のように $time==0 の結果もプリントされます。この事実から、p1 と p2 の実行開始順序が問題である事が確認できました。

```
time a b {co,sum}
@ 0: 0 0    0
@10: 0 1    1
@20: 1 0    1
@30: 1 1    2
```

勿論、この解決法は望ましいとは言えませんが、プロセスの実行順序が検証結果の差異をもたらしている事実を確認するには十分な例であると思います。正しい解決法は幾つか存在しますが、比較的簡単な解決法を以下に紹介しておきます。以下では、p1 が p2 よりも先に実行を開始すると仮定して問題点を解消する手順を解説します。

p1 の内部の実行を p2 よりも後に実行させれば良いので、p1 の for ループ部分を fork-join で囲めば問題は解消します。例えば、以下のように p1 を書き換えれば良いです。

```
initial begin: p1
    fork
        for( int i = 0; i < 4; i++ ) begin
            {a,b} = i;
            #10;
        end
    join
end
```

fork で子プロセスとして生成されるので p2 よりも後に実行を開始する

fork で生成された子プロセスは、直ぐには実行されずにスケジューリングされるだけです。join で p1 が待ち状態に入りますが、子プロセスの前に p2 が既にスケジューリングされているので、p2 が次に実行を開始します。実行を開始すると、p2 は @(a,b) を実行するので

イベント待ち状態になります。そして、今度は fork の子プロセスの実行が開始して、{a,b} に 0 がセットされます。この時点では、p2 における @(a,b) のイベント待ちが有効になっているので、$time==0 におけるイベント待ちが解除されてプリントされます。
■

参考 1-1

例 1-1 で解説した問題は、テストベンチ自身の問題であるので組み合わせ回路には関係がありません。したがって、シーケンシャル回路に対しても、同様の問題が発生します。すなわち、DUT をドライブするプロセスと DUT からのレスポンスをサンプリングするプロセスの実行順序を正しく管理しないと、$time==0 における検証は正しく行えない可能性があります。
□

DUT からのレスポンスを適切なタイミングで取得しなければ DUT を正しく検証する事ができません。そして、レスポンスを正しく取り出すためには、DUT の出力が安定した後に、その値を取り出さなければなりません。次に、取得するタイミングの重要性を解説します。

1.2 DUT とコレクター*

DUT からの出力をサンプリングするプロセスは、一般的に、コレクターと呼ばれます。本書では、その用語を踏襲する事にします。

組み合わせ回路は Active 領域で動作し、シーケンシャル回路は Active 領域と NBA 領域で動作します（表 1-1）。

表 1-1 デザインが実行するシミュレーション領域

回路種別	シミュレーション領域
組み合わせ回路	Active
シーケンシャル回路	Active、NBA

先ず、組み合わせ回路とコレクターのタイミング問題を考察する事から始めます。

1.2.1 組み合わせ回路とコレクター

組み合わせ回路を always プロシージャで以下のように記述したと仮定します。回路の機能は重要ではないので処理を省略しています。

```
module comb_design(input a,b,output logic z);
always @(a,b)
    z = ...;
endmodule
```

この記述によれば、入力信号 a または b の何れかに変化が起これば z に新しい計算値が設定されます。

しかし、ソースファイルの他の場所にも 同じような always @(a,b)[1] があれば、その always プロシージャが comp_design の always よりも先に実行を開始する可能性があります。その場合には、入力信号 a または b の何れかに変化が起こっても、z には直ぐに新しい計算値はセットされずに、暫くしてから z の値が更新されます。言い換えると、回路の入力信号に変化が起こっても、出力値 z が安定するまでは暫くの時間を要すると考えるのが妥当です。その時間幅は、Active 領域と呼ばれています（図 1-1）。なお、シミュレーション領域に関しては、第 2 章で詳しく解説します。

[1] always @(a,b) だけでなく、always @(…,a,…)、always @(…,b,…) 等も含まれます。

3　タイミング制御

既に紹介したように、SystemVerilog ではタイミング制御として以下の機能を備えています。

- ディレー制御（#と##オペレータ）
- エッジセンシティブイベント制御（@expr）
- レベルセンシティブイベント制御（wait 文）

タイミング制御機能を使用しないと、シミュレーションは$time==0 で終了します。また、タイミング制御を使用しても、イベント待ちとイベント解除のタイミングがずれるとイベント待ちが解除されないという現象が起こります。したがって、クリティカルなタイミング状況においては、エッジセンシティブとレベルセンシティブを確実に使い分ける必要があります。エッジセンシティブイベントの発生は瞬間であるのに対して、レベルセンシティブイベントが発生した状態は一定期間継続します。例えば、以下の記述を仮定します。

```
logic  a;

initial begin: p1
    a = 1;
end

initial begin: p2
    @(posedge a) $display("@%0t: a = %b",$time,a);
end
```

この記述例にはエッジセンシティブイベント制御が使用されていますが、イベント待ちが有効になる時期とイベントを起こさせている文の実行時期が不安定であるため、結果は実行順序に依存します。もし p1 が p2 よりも先に実行すれば、$display システムタスクは$time==0 においてプリントしません。しかし、p2 が先に実行すれば、イベント待ちが有効になってから a の値が 1 になるので、$display システムタスクは$time==0 においてプリントされます。

上記の例は、エッジセンシティブなイベント制御に固有の問題です。しかし、レベルセンシティブイベント制御ではそのような問題はありません。例えば、以下に示す記述例を仮定します。

```
logic  a;

initial begin: p1
    a = 1;
end

initial begin: p2
    wait( a ) $display("@%0t: a = %b",$time,a);
end
```

p1 が p2 よりも先に実行すれば a == 1 なので、wait 文の条件は満たされて、$time==0 において a の値がプリントされます。逆に、p2 が p1 よりも先に実行すると、wait 文はイベントを待つ状態に入り、その後に a が 1 にセットされると、wait 文が解除されて、a の値がプリントます。結局、p1 と p2 のどちらが先に実行しても同じ結果が得られます。

参考 3-1

これらの例が示すように、タイミングがクリティカルな場合には、つまり、イベント解除とイベント待ちが同時刻に起こる場合には、レベルセンシティブイベント制御の方が確実である事が分かります。ただし、レベルセンシティブイベント制御には、イベント待ちが解除さ

れた後に、その状態をリセットするタイミングが非常に難しいという問題があります。例えば、wait(a)が解除された後に a の値をリセットするタイミングを正確に求めなければなりません。この目的に即した SystemVerilog 機能は名前付きイベントの triggered メソッドです。

triggeredメソッドはエッジセンシティブイベント制御をレベルセンシティブイベント制御に変換する機能を持ちますが、変換機能はシミュレーション時間が進行すると自動的にリセットされます。
□

3.1 概要

タイミング制御機能の差異を表 3-1 に整理しておきます。

表 3-1 タイミング制御機能の差異

タイミング制御	特徴	イベントの発生	特異条件
ディレー	ディレー制御をしているプロセスは実行を中断し、指定したディレー後に実行を再開するようになります。実行再開に関しては、他のプロセスに依存しません。つまり、シミュレーションが進行して、指定した時刻に到達すれば、プロセスは実行を再開するようにスケジューリングされます。	指定した時間が経過すれば、自動的にイベント待ちが解除されます。	時間の変化
エッジセンシティブ	イベント待ち制御をしているプロセスは実行を中断し、指定したイベントが発生するまで待ちます。実行再開に関しては、他のプロセスに依存します。	エッジセンシティブイベントに指定されている式の値の変化があれば、イベントが発生したと判断されます。	LSB の変化 値の変化
レベルセンシティブ		レベルセンシティブイベントに指定されている式の値が指定した条件を満たせば、イベントが発生したと判断されます。	値の状態

参考 3-2

エッジセンシティブな表現は@(...)のように指定されますが、@a と@(posedge clk)の形式では意味が異なります。以下のような環境を仮定して相違を解説します（表 3-2）。

```
logic [7:0]   a;
logic     clk;
event     ev;
```

4 プロセス

initial と always プロシージャは、シミュレーション開始時に自動的に起動される標準プロセスです。これ以外のプロセスを生成するためには、fork 機能を使用しなければなりません。SystemVerilog の fork 機能には以下の三つのバリエーションがあります。

- fork-join
- fork-join_any
- fork-join_none

何れの機能も並列処理プロセスを子プロセスとして生成しますが、通常、検証作業で使用するのは、fork-join_none です。理由は、自分自身を中断せずに子プロセスを生成できるからです。しかも、生成した子プロセスをいつでも実行可能にすることができる利点があります。fork-join と fork-join_any は子プロセスの実行開始に関して柔軟性に欠けます。

柔軟性の一例として、fork-join_none で fork-join のような動作を実現するには以下のようにします。

```
fork
...
join_none
wait fork;
```

このようにすると、fork-join_none で生成した全ての子プロセスが終了するのを待つことができます。

4.1 fork 機能

always プロシージャは繰り返して実行する性質を持つので、このプロシージャ内に fork 機能を記述する事は非常に考え難いのですが、initial プロシージャ内に fork 機能を組み込むのはごく普通です。一般的には、タスク内で fork 機能を使用します。更に、fork-join_none の場合には、ファンクション内でも使用する事ができます。

例えば、ファンクション内では、以下のように fork-join_none を使用します。この場合には、void ファンクションが使用されていますが、値を戻すファンクション内で使用する場合には注意が必要です。

```
function void do_something();
...
fork
    ...
join_none
...
endfunction
```

生成されたプロセスは、スケジューリングされますが自動的には実行を開始しません。親プロセスが終了、またはイベント待ちになるまで子プロセスの実行は開始しません。したがって、子プロセスの実行開始は多少遅れます。しかし、$time==T で生成された子プロセスは、必ず、$time==T で実行が開始されます。

fork-join と fork-join_any では、join または join_any を実行した時点で親プロセスが待ちの状態になるため、スケジューリングされた子プロセスの実行が自動的に開始します。したがって、fork-join と fork-join_any の場合には、子プロセスの実行開始に関して特別な配慮は必要ありません。

しかし、既に述べたように、fork-join_none は柔軟性を備えている事と、使用面で他の fork 機能よりも高度な制御が必要であるため、本章では、主として、fork-join_none を取り扱います。

例 4-1　fork-join_none の使用例

以下に、最も典型的な fork-join_none の使用法を示します。記述にあるように、join_none の後の適当な時期に親プロセスはイベント待ちに入る必要があります。以下の記述では、二つの子プロセスを生成していますが、親プロセスが実行を中断するまで、子プロセスの実行は開始していません。

```
module test;

initial begin
    fork
            print("proc1");
        #10 print("proc2");
    join_none

    print("main");
    #50 $finish;
end

function automatic void print(string msg);
    $display("@%2t: %s",$time,msg);
endfunction

endmodule
```

#50 のディレー制御に遭遇すると、親プロセスがイベント待ちになる

親プロセスが、fork-join_none で二つの子プロセスを生成した後、ディレー#50 に遭遇すると、親プロセスは実行待ちの状態になります。この時点で二つの子プロセスは実行を開始します。

最初の子プロセス print("proc1")は実行開始後すぐ終了します。二番目の子プロセスは、#10 のディレーによりイベント待ちになり、その後、print("proc2")が呼ばれます。

実行すると以下のような結果を得ます。この結果から、親プロセスが実行を中断するまで、子プロセスが実行を開始しない事を確認できます。

```
@ 0: main
@ 0: proc1
@10: proc2
```

■

例 4-1 では、親プロセスの実行を中断するためにディレー#50 を使用しましたが、通常は、ディレー#0 を使用して、汎用的な処理を記述します。したがって、fork-join_none を使用する一般的な記述法は以下のようになります。

```
fork
...
join_none
...
#0;
```

この記述法は、生成された子プロセスが実行開始をした後、すぐに親プロセスも実行を再開する場合に有効な手法です。しかし、この方法では子プロセスの実行がどの程度進んでいる

7　検証法

本章では、これまでに解説した検証に関する重要な事項を体系的にまとめます。

7.1　検証におけるタイミング

組み合わせ回路でもシーケンシャル回路でも共通にいえる事ですが、DUT をドライブするタイミングと DUT の出力をサンプリングするタイミングに注意しないと正しい検証を行えない場合があります。組み合わせ回路を例にとり、検証におけるタイミングの重要性を示します。先ず、検証するタイミングの基本的な原則を以下にまとめておきます。

SystemVerilog はイベントドリブン方式によるシミュレーション原理を採用しているので、信号に関するイベント待ちが有効になった後にその信号のイベントが起こらなければなりません。例えば、`@(a)` のイベント待ちが有効になった後に `a` の値が変化すれば、イベント待ち `@(a)` は解除されます。しかし、順序が逆になるとイベント待ちは解除されません。

次に、簡単な例で問題点を明確にします。

例 7-1　実行順序に依存する検証例

実行順序に依存する検証例を以下に示します。この例では、組み合わせ回路として AND 回路を `always` プロシージャでモデリングしています。そして、AND 回路をテストするために、`$time==0` で入力 `a` と `b` に値を設定しています。

```
module test;
logic  a, b, z;

initial begin
    a = 1;
    b = 1;
end

always @(a,b)
    z = a&b;

endmodule
```

AND 回路の入力値を設定

AND 回路のモデリング

しかし、このテストベンチは実行順序に依存するため正しい検証法とは言えません。例えば、`initial` プロシージャが `always` プロシージャよりも先に実行すれば、イベント待ち `@(a,b)` が有効になる前に `a` と `b` の値が変化するので、イベント待ち `@(a,b)` は解除されません。逆に、`always` プロシージャが先に実行すればイベント待ち `@(a,b)` は解除されます。つまり、検証結果は実行順序に依存する事がわかります。

次に、解決法を解説します。モデリングを変える事はできないので、検証法を変更しなければなりません。問題は、イベント待ち `@(a,b)` が有効になる前に、`a` と `b` に対する値の設定が実行してしまう点です。順序を逆にするためには、`a` と `b` への設定を遅らせれば良いので、以下のようにすれば問題は解決します（表 7-1）。

表 7-1　実行順序に依存しない検証法

解決法 1	解決法 2

`initial begin`
`    a <= 1;`
`    b <= 1;`
`end`
	`initial begin`
	`    fork`
	`        begin`
	`            a = 1;`
	`            b = 1;`
	`        end`
	`    join_none`

	end
a と b の値の設定は NBA 領域で実行するので、それまでにはイベント待ち@(a,b)が有効になっています。	fork で生成されるプロセスは、Active 領域のプロセスの実行が終了した後または実行が中断した後に行われます。したがって、イベント待ち@(a,b)が有効になってから、a と b に値が設定されます。

■

検証要素の基本はドライバーとコレクターなので、これらの検証要素を開発する時に正しいタイミングで検証すれば良い事になります。以下では、組み合わせ回路とシーケンシャル回路の正しい検証法を紹介します。

7.2 組み合わせ回路の検証法

組み合わせ回路では、全ての命令は Active 領域で実行するので、以下の原則に従えば正しい検証を行えます。

① ドライバーとコレクターのプロセスを fork で生成する。
② コレクターは、#0 を使用して Inactive 領域で DUT の出力をサンプリングする。
③ コレクターの準備が完了した後にドライバーは DUT をドライブし始める。

これだけのルールを守れば良いです。SystemVerilog のルールにより以下の事が成立します。

- ドライバーが起動する時には、DUT の連続代入文と always が既に実行を開始しています。したがって、ドライバーはいつでも DUT をドライブできます。SystemVerilog（寧ろ、論理合成）のルールにより、always の先頭にセンシティビティリスト@(...)がある事に注意して下さい。つまり、連続代入文も always も実行を開始すると直ぐにイベント待ち状態に入ります。
- コレクターが起動する前に DUT が実行を開始していますが、同じ$time==0 でコレクターも実行を開始します。つまり、DUT のイベント待ちが解除される以前にコレクターの実行が開始されています。もし、$time==0 で DUT が出力を生成しても、それは Active 領域で生成するので、Inactive 領域で出力を監視しているコレクターが見逃す筈はありません。
- コレクターのイベント待ちが有効になった後であれば、ドライバーはいつでも DUT をドライできるのは明らかです。

例 7-2 組み合わせ回路の検証例

例 7-1 で紹介した検証法を拡張して検証ルールの効果を確認します。先ず、以下のようにドライバーとコレクターのプロセスを fork で生成します。ただし、コレクターがドライバーよりも先に実行するようにプロセス制御をします。以下の例では、#0 を使用してプロセス間の同期を取っています。こうすると、コレクターが実行を開始するまではドライバーは DUT をドライブしません。

```
module test;
logic  a, b, z;

initial begin
   fork
      driver;
      collector;
   join_none
end
```