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-002-2021
Document Revision: 1.1, 2021.12.15
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

資料単価：3200 円（金額は消費税を含んでいません）
連絡先：contact.us@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。ただし、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。

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
```

SystemVerilog が難しいのは、機能が広範囲にわたる事を別として、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。この規約により、宣言時の初期化内にはモジュールやインターフェースのインスタンス等の実行時に関わる情報を使用する事ができない事が分かります。つまり、宣言時の初期化では環境依存性と初期化順序を考慮しなければならないと言えます。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評価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す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する事ができます。

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 初心者にお勧めします。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。

目次

本書で使用する略語一覧

略語	定義
DUT	Design Under Test、または、Device Under Test
EDA	Electronic Design Automation
FSM	Finite State Machine
IPC	Inter-Process Communication
LHS	Left Hand Side
LRM	Language Reference Manual、すなわち、IEEE Std 1800-2017
LSB	Least Significant Bit
RHS	Right Hand Side
RTL	Register Transfer Level
TLM	Transaction Level Modeling
UVM	Universal Verification Methodology

1　シミュレーション

本章は、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勧めます。

1.1　ハードウェアの記述と並列処理*

SystemVerilog 言語仕様 IEEE Std 1800-2017（以下、LRM と略称）は、言語の文法および機能を厳密に規定していますが、機能をどのように実装すべきかに関しては EDA ベンダーに委ねられています。それにもかかわらず、同じデザインを異なるシミュレータで実行しても、同一の結果が得られます。ここで、「同一」とは、「実行環境に依存する結果の差異を除外する」という意味です。例えば、下記の記述例では、二つの並列処理プロセス p1 と p2 が処理を同時に開始します。p1 と p2 のどちらが先に実行するかは分からないため、実行順序に伴う結果の差異を度外視しなければなりません。

```
initial begin: p1
   $display("@%0t: p1 started",$time);
   // ...
end

initial begin: p2
   $display("@%0t: p2 started",$time);
   // ...
end
```

p1 と p2 のどちらが先に実行するか分からない

しかし、シミュレーション結果を目視ではなく自動的に検証するとなると、全く別の話になります。つまり、シミュレーション結果が実行環境に依存しないように検証コードを準備しなければなりません。例えば、前記の記述例を以下のように書き換えて実行動作を解析する事にします。

```
event  p1_ready;

initial begin: p1
   ->p1_ready;
   $display("@%0t: p1 started",$time);
   // ...
end

initial begin: p2
   wait( p1_ready.triggered );
   $display("@%0t: p2 started",$time);
   // ...
end
```

p1 の内部は p2 の内部よりも先に実行する

この場合、実行すると以下に示すようなメッセージで始まります。

```
@0: p1 started
@0: p2 started
...
```

仮に p1 が p2 よりも先に実行を開始すれば、上記のようにメッセージがプリントされる結果は明らかです。では、逆の場合にはどうなるかを解析してみます。SystemVerilog の規約によれば、プロセスはディレーやイベント待ちにより実行が中断するまでは実行権を持ち続けます。p2 が先に実行を開始しても、wait 文により、p2 は p1 が実行を開始するのを待ちます。p1 が実行を開始すると、->p1_ready は p1 を中断しないので、p1 はそのまま続行します。

したがって、上記のようにメッセージがプリントされます。以上から、p1 と p2 の実行順序に関わらず、同じ結果が得られます。

参考 1-1

上記の例では、イベント解除は実行をブロックせずに実行を継続すると仮定しました。例えば、p2 が先に実行した時、以下のように p1 は p1_ready のイベント解除を行い、そのまま実行を継続して、$display システムタスクでメッセージをプリントすると仮定をしました。

```
initial begin: p1
    ->p1_ready;
    $display("@%0t: p1 started",$time);
    // ...
end
```

しかし、p1 が p1_ready のイベント解除を行った後に、p1 の実行を継続する前に、p2 の実行を再開する処理系も間違いではありません。

```
initial begin: p1
    ->p1_ready;   ----------------------------①
    $display("@%0t: p1 started",$time);
    // ...
end

initial begin: p2
    wait( p1_ready.triggered );
    $display("@%0t: p2 started",$time);
    // ...    ----------------------------②
end
```

p1_ready のイベント解除後、実行制御が p2 に移行する処理系も正しい

この場合には、以下に示すようなメッセージで始まります。

```
@0: p2 started
@0: p1 started
...
```

イベント解除はイベント待ちのプロセスを解除しますが、イベントを解除するプロセス自身の実行を中断するわけではないので、本書では、イベント解除をするプロセスはイベント解除後も実行を継続するアプローチを取ります。並列処理言語の環境では、このような競合状態は常に発生するため、実行結果の観察には十分な吟味が必要です。
□

もちろん、実行順序依存性を回避するには他の方法もあるので、ここで紹介した方法は一例に過ぎません。しかし、ここでの重要な点は、実行順序に依存する記述の有無をコードの開発者が認識できるか否かの問題です。もし認識する事ができなければ対策すら立てる事ができません。

SystemVerilog はハードウェアを記述するための言語であり、ハードウェアは、元来、並列処理を基本としています。したがって、SystemVerilog で記述したデザインは並列処理の集まりと言えます。デザインがハードウェアの仕様通りに動作するためには、実行環境に依存しない一定の結果を生じなければなりません。そして、「一定の結果」を得るためにエッジセンシティブ、およびレベルセンシティブな制御を使用して並列処理プロセス間の同期を取る必要があります。

タイミングを取る意味においては、ディレー（# と##オペレータ）制御もありますが人為的です。何故なら、ディレーはプロセス自身の実行を遅らせるだけであり、他のプロセスとの同期を取る目的に使用する事ができません。つまり、ディレーはハードウェアを実現する手段としての機能ではありません。ディレーは、あくまでもテストベンチにおける検証手段に過ぎません。

1.2　シミュレーション時間の進行*

通常の SystemVerilog 実行命令では、シミュレーション時間は進行しません。例えば、以下の記述に示す命令が$time==T において実行されたとすると、実行終了後も$time==T です。

```
tmp = a & b;
q <= tmp;
```

$time==T において実行された命令が、T よりも後の$time で終了するためには、以下の何れかの機能が使用されなければなりません。本書では、これらの機能を総称してタイミング制御機能と呼ぶ事にします。

- ディレー制御（# および ## 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）

これらのタイミング制御機能が全く使用されない場合、シミュレーションは$time==0 で終了します。例えば、以下のような単純なループでは、$time は進行せずに同じ命令を繰り返すだけで意味を持ちません。

```
initial forever clk = ~clk;
```

この場合には、以下のように訂正する必要があります。

```
initial forever #10 clk = ~clk;
```

しかし、以下のように訂正しても全く意味がありません。

```
initial forever #10;
```

何故なら、この命令はプロセス自身を定期的に実行しているだけであり、他のプロセスの活動には影響を与えていないからです。他のプロセスの実行に影響を与えるためには、信号値の変化を発生しなければなりません。

タイミング制御機能は、表 1-1 にまとめられています。ディレー制御は命令を必ずスケジューリングしますが、イベント制御はイベントが発生するまでは命令をスケジューリングしません。

表 1-1　タイミング制御機能

タイミング制御機能	機能	効果
ディレー #n	プロセスの実行を将来の確定した時刻にスケジューリングする機能です。	ディレー制御に遭遇した時刻を$time とすると、プロセスが$time+n に再開するようにスケジューリングします。
サイクルディレー ##n		このサイクルディレーに遭遇すると、n クロックサイクル後に実行が再開します。

2　シミュレーション実行モデル

SystemVerilog では、時刻 T におけるシミュレーションは幾つかのスケジューリング領域に分かれて時系列的に実行されます。ただし、それぞれの領域を順に実行するのではなく、新たなイベントが発生すると以前の領域に戻りシミュレーションを繰り返すという複雑なシミュレーション方式を取っています。例えば、NBA 領域におけるノンブロッキング代入文　q <= d の実行により、q の値に変化が起これば、q に依存する信号の更新を行うために以前の領域に戻り再計算を行わなければなりません。

しかし、この方式により複雑なイベント制御の流れを論理的に正しい順に処理する事ができます。全体的にはやや複雑な実行の流れとなりますが、ハードウェア仕様を満たすデザインを正確に記述し、かつ正しく検証するためには、完全に理解しなければならない SystemVerilog のシミュレーションルールです。

この SystemVerilog のシミュレーションルールにより、信号値の変化か起こるタイミングを正確に知る事ができるようになりますが、実際に信号値を確認するための検証動作を準備する事は別問題です。むしろ、その検証動作を記述する方が難しい課題となります。これらの状況を踏まえて、SystemVerilog 記述が提起するタイミング問題だけでなく、SystemVerilog シミュレーションルールが検証を可能にしている事を詳しく解説します。先ず、簡単な例を示して、シミュレーション結果の検証法を解説する事から始める事にします。

2.1　背景

先ず、簡単な例によりタイミングの問題を提起します。もし以下のような記述をすると、三つの代入文が同時に実行を開始するため競合状態が発生し、変数 a に設定される値は予想が付きません。明らかに、結果は実行順序に依存します。

```
fork
    a = 1;
    a = 2;
    a = 3;
join
```

しかし、a に値を設定するタイミングを以下のように調節すると実行環境に依存せずに常に一定の結果を生じます。

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

この場合には、変数 a は順に、1→2→3 と変化して行きます。実行環境に依存せずに、なぜ一定の結果を得られるかは、SystemVerilog の規約の賜物です。それを理解するために、上記のコードが実行するスケジューリング領域を注釈として以下のように追加します。

```
fork
    a = 1;        // Active 領域
    #0 a = 2;     // Inactive 領域
    a <= 3;       // NBA 領域
join
```

最初の代入文は、Active 領域で実行するという意味で、他の文も同様に、対応する領域で実行します。そして、SystemVerilog の規約によれば、これらの実行の順序は図 2-1 のようになります。

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の発生は一定期間継続します。

例えば、以下の記述を仮定します。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。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。

```
logic  a;

initial begin: p1
   a = 1;
end

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

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、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、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。詳細は、文献[4]を参照して下さい。
□

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 LSB が指定した条件を満たせば、イベントが発生したと判定されます。	LSB 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。	値の変化

参考 3-2

以上の解説から、ディレーは自身のプロセス制御としてしか意味がない事が分かります。つまり、ディレーは他のプロセスには一切関係がないので、プロセス間の同期を取る目的に使用する事はできません。一方、エッジセンシティブおよびレベルセンシティブ制御のイベント待ちは、他のプロセスによるイベント待ちの解除を前提とするため、プロセス間の同期を取る目的に使用する事ができます。
□

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イベント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、子プロセスの実行に関して特別な配慮は必要ありません。

しかし、既に言及したように、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。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

6 クラス

近年の検証環境はクラスを使用して構築されます。クラスのオブジェクトを何時でも作り出すことができるので、ダイナミックに検証環境の構成を変更する事ができる利点があります。一方、モジュールインスタンスは、コンパイル時に決定していなければならない情報であるため柔軟性に欠けます。本章では、クラスを検証環境構築の手段として使用する場合を想定して、クラスの使用法を解説します。

クラスには多くの機能と重要な性質がありますが、検証環境を構築する観点から以下に示す機能を正確に使用しなければなりません。

- static と automatic 変数
- new コンストラクタ
- virtual インターフェース

先ず、これらの機能の使用上の注意点から解説する事にします。

6.1 static と automatic 変数

モジュールで定義したネットや変数は、原則として、static です。つまり、それらのオブジェクトは、シミュレーション実行中に常に存在します。

例えば、以下の記述において、変数 clk は static であり、常時、存在します。したがって、以下のようにクロック生成を繰り返すことが出来ます。

```
module test;
logic  a, b;
bit    clk;
// ...
initial begin
   a = 0;
   b = 0;
// ...
end

initial forever #10 clk = ~clk;

endmodule
```

clk は常に存在するので、常時、参照できる

一時的にしか存在しない変数は automatic であり、使用されている期間を過ぎると消滅します。クラスに定義された変数は、原則として、automatic で、クラスのオブジェクトが存在する間のみ使用可能です。ただし、メソッド内に定義されている変数は、そのメソッドが実行中の時のみ使用可能で、メソッドが終了すると消滅します。

例えば、以下の simple_item_t クラスに定義されている port と addr は automatic ですが、このクラスのオブジェクトが存在する限り有効なランダム変数です。つまり、これらの変数はクラスオブジェクト毎に存在します。

```
class simple_item_t;
static int          count;
rand logic [3:0]    port;
rand logic [15:0]   addr;

constraint C_PORT { port inside {[0:5]}; }
constraint C_ADDR { addr[1:0] == 2'b00; }

// ...
```

```
endclass
```

一方、変数 count は static と宣言されているので、simple_item_t クラスのオブジェクトとは無関係に存在します。count は、シミュレーション開始する前に確保され、終了するまで有効になります。

参考 6-1

クラスの static 変数は、クラスのオブジェクトと無関係に存在します。しかも、クラス外からもこの変数を参照する事ができます。通常は、同じクラスタイプが共有する情報、または同じクラスのタイプに共通する情報を static 変数として定義します。
□

6.2 new コンストラクタ

クラスのオブジェクトを作る時には、必ず、コンストラクタが呼ばれます。たとえ、クラスにコンストラクタが定義されていなくてもコンストラクタを呼ぶ事ができます。これは、コンパイラーが、以下のような空のコンストラクタを定義しているからです。

```
function new;
endfunction
```

クラスに定義した変数は、ランダム変数を除くと、初期化が必要です。標準値以外の初期化が必要な場合には、コンストラクタで初期値を設定するのが一般的です。

例えば、以下の記述例は object_t クラスの定義を示していますが、変数 name と identifier はクラスオブジェクト毎に存在し、異なる値が設定されなければなりません。しかも、オブジェクトが生成されると、それらの変数が使用可能でなければならないので、コンストラクタで初期化されなければなりません。

```
class object_t;
static int    count;
string        name;
int           identifier;

function new(string name);
   this.name = name;
   identifier = ++count;
endfunction

endclass
```

参考 6-2

object_t クラスでは、static なプロパーティ count を使用して、identifier を初期化しています。したがって、count はクラスのオブジェクトが作られる以前に使用可能な状態になっていなければなりません。シミュレーションが開始する前に、static なプロパーティが初期化されますが、具体的に何時それが行われるかは SystemVerilog では明確にされていません。このため、クラスのオブジェクトをシミュレーション開始以前に作ると不安定な検証環境が構築される事になります。

例えば、以下の記述では、obj の初期化をトップモジュールの外で行っているので、クラスのオブジェクトはシミュレーションが開始する前に作成されますが、object_t::count の準備ができてから行われる保証はありません。

7　式の計算精度

7.1　概要

C/C++では、演算結果の精度は殆ど自明であり悩む事はないと思われます。しかし、SystemVerilog では、オペランドのビット長は任意の長さを持つため、演算結果のビット長の計算は自明な問題ではありません。例えば、以下のような記述を引用して考察します。

例 7-1　計算精度が式の記述に依存する例（[1]）

同じ演算記述でも、他のオペランドに依存して精度が決定される例を以下に示します。

```
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 ビットで行われます。したがって、計算中にオーバーフローが発生しても無視されるため、sumA には正しい結果が求まりません。一方、sumB を計算するための加算 a+b は、17 ビットで行われます。したがって、sumB の計算式ではオーバーフローが考慮されているので、正しい計算結果が得られます。
■

このように、記述の仕方により計算結果が全く異なる事が分かります。sumA の計算方式が意図的に行われているのであれば問題はありませんが、計算精度に関する知識が不十分であるために発生した結果であるならば SystemVerilog の演算の仕組みを十分に理解する必要があります。本章では、計算精度を制御する仕組みを解説します。その仕組みを理解する事により、予期しない事態を未然に防ぐことが出来ます。

7.2　計算精度と演算精度

以下での解説を厳密にするために、計算精度と演算精度を定義しておきます。演算精度は、演算自身で決定される演算結果のビット長です。一方、計算精度は、演算が式の中に使用された時に解釈される演算結果のビット長を意味します。例 7-1 に対しての演算精度と計算精度の意味は表 7-1 のようになります。

表 7-1　計算精度と演算精度の例

式	演算	演算精度	計算精度
sumA = a + b;	a+b	16	16
sumB = a + b;	a+b	16	17

既に解説をしたように厳密な計算結果を得るためには、式に使用されているネットや変数を正しく宣言しなければなりません。

7.3　演算精度

C/C++では、演算 100*200 は 32 ビットで行われ、結果は 32 ビットで求まります。確かにリテラル定数を使用すれば、SystemVerilog でも同様に 32 ビット[1]で行われますが、変数同士の乗算となると 32 ビットとは限りません。先ず、演算精度のルールを表 7-2 に示します。表 7-2 においては、L(i)はオペランド i のビット長を意味します。

ここで使用している L(i)は、システムファンクション$bits を意味します。したがって、a*b の演算精度は、max($bits(a),$bits(b))として求める事ができます。

[1] SystemVerilog では、整数リテラルは 32 ビット以上であり 32 ビットとは限りません。ここでは、32 ビットと仮定しています。

表 7-2　主要なオペレータと演算精度

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))

参考 7-1

オペレータ（>>、<<、**、>>>、<<<）の右オペランドは演算精度に影響を与えません。
□

例 7-2　演算精度計算例

以下の式に使用されている加算 a+b の演算精度は max($bits(a),$bits(b))に等しいので、5 となります。左辺も 5 ビットなので、5 ビットで計算されます。

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

assign sum = a+b;
```

加算 a+b は 5 ビットで計算される

■

演算精度はオペランドで決定されますが、式全体の精度にしたがい演算精度は調節されます。次に、その調節する仕組みを解説します。

7.4　計算精度

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。例 7-1 に紹介した記述例から引用して解説します。便宜のために以下に、再度、記述例を示します。

```
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
```

変数 a と b は 16 ビットです。sumA は 16 ビットなので、sumA を求める加算 a+b は 16 ビットで行われます。一方、sumB は 17 ビットなので、sumB を求める加算 a+b は 17 ビットで行われる事になります。したがって、sumB にはオーバーフローした場合の考慮がされています。

＋オペレータだけでなく、他のオペレータに関しても同様な事が言えます。例えば、a*b は 16 ビットの演算精度ですが、以下のようにすると a*b は 17 ビットで行われます。

```
sumB = a*b;
```

8　補足

本章では、SystemVerilog が規定している主な規則と特殊な使用法の注意事項を整理しておきます。

8.1　SystemVerilog の規則

8.1.1　名称の長さ

名称は、オブジェクトを一意的に示すために与える名前です。名称として許される最大長は使用するソフトウェアツールに依存しますが、最大長は少なくとも 1024 文字である事が SystemVerilog で規定されています。

8.1.2　リテラルのビット長

8.1.2.1　'0、'1、'x、'z

SystemVerilog のリテラル'0、'1、'x、'z 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、これらのリテラルは符号なしです。

例 8-1　可変長リテラルの効果

以下のように `state` を宣言すると 1024 ビットの packed アレイが準備されます。そして、`state` の全てのビットは 1 で初期化されます。

```
logic [1023:0]   state;
state = '1;
```

■

8.1.2.2　サイズ指定なしのリテラル

SystemVerilog では、サイズを指定しない数値リテラル（100、'habcd、'b1010、等）は、少なくとも 32 ビットの大きさを持たなければならないと規定しています。実際のビット数は使用するソフトウェアツールにより異なります。

例 8-2　状況に依存する判定例

以下の記述では、リテラル 8 が 32 ビットであると仮定していますが、絶対的に正しい判定法とは言えません。もし 8 が 64 ビットであれば、下記の判定が真になる事はありません。

```
logic [3:0]   a;

initial
    ...
    if( {a,8} == 36'hf_0000_0008 )
        ...
end
```

■

8.2　注意が必要な記述方式

8.2.1　ビットセレクトとパートセレクト

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 8-3　パートセレクトの例

以下の例において、変数 b には符号が付いていますが、`b[7:0]`は符号なしです。