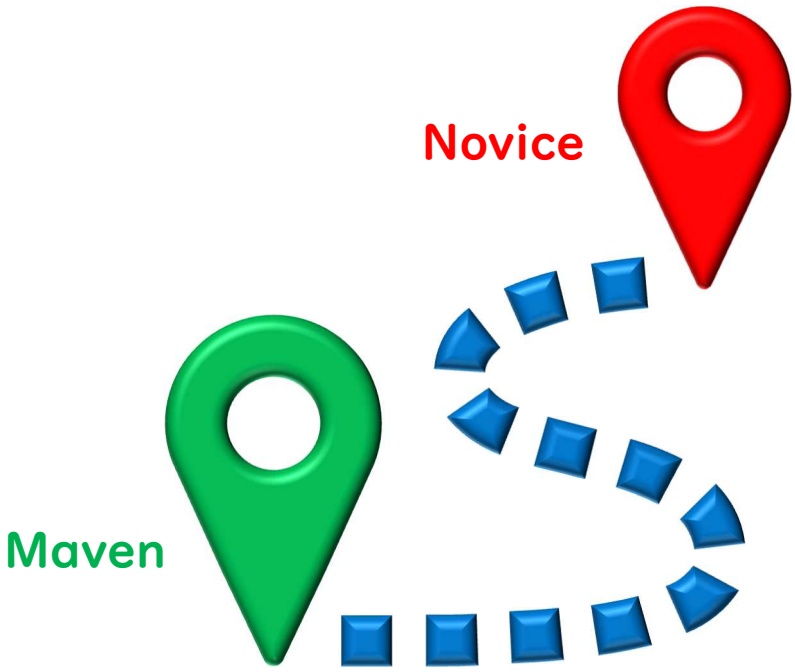
Artgraphics

SystemVerilog による効果的実装技術

Document Identification Number: ARTG-TD-004-2024
Document Revision: 1.1, 2024.11.22
アートグラフィックス
篠塚一也



SystemVerilog による効果的実装技術

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

Effectual Implementation using SystemVerilog

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

はじめに

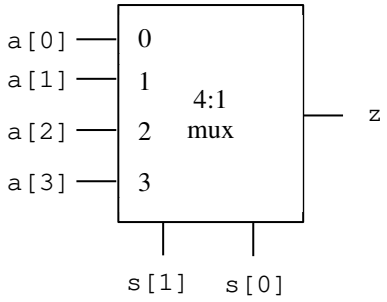
本書は、SystemVerilog に関する基礎知識を持っている人を対象にして、SystemVerilog の機能を効果的に使用するための技術を解説した技術資料です。SystemVerilog は豊富な機能を備えていますが、それぞれの機能を有効に結合して初めて効果的な使用法が確立します。古くから既に確立されている標準的な記述法でも唯一の方法とは言えません。時として、それらの手法を吟味し直すと全く異なる表現法につながる場合があります。

例えば、式の右辺に使用されたインデックス指定のビットセレクトは、一般的に、マルチプレクサに合成されますが、ビットセレクトを使用しなくても、同等の表現ができます。以下に示すように、シフトオペレータ（>>）を使用してもインデックス指定のビットセレクトを表現できます。

```
module design1(
    input [3:0] a,[1:0] s,
    output z);
assign z = a[s];
endmodule
```

```
module design2(
    input [3:0] a,[1:0] s,
    output z);
assign z = a>>s;
endmodule
```

s	a[s]
0	a[0]
1	a[1]
2	a[2]
3	a[3]



何故なら、(a>>s)では、(s==0)であればa[0]、(s==1)であればa[1]、(s==2)であればa[2]、(s==3)であればa[3]がzに設定されるからです。しかも、zが4:1マルチプレクサで表現される事は明白です。

このように、既に確立されている実装法にとらわれずに、現在用いられている手法を吟味し直すと、他の実装法に辿り着く機会に恵まれる可能性が高まります。時として、新発見された手法が画期的な性能改善につながる可能性もあります。本書は、そのような好機を作り出すための“ひらめき”を養うための技術資料です。

また、慣習的におこなわれている記述法を見直す事も必要です。同じ機能を実現できる二つの方法があれば、汎用性・簡潔性・性能などの観点から比較を行い、望ましい記述法を選択する努力は大きな改善に結びつきます。下記は、その一例です。

慣習的な記述法	オペレータを活用する記述法
always @(code) case (code) 0: data = 8'b0000_0001; 1: data = 8'b0000_0010; 2: data = 8'b0000_0100; 3: data = 8'b0000_1000; 4: data = 8'b0001_0000; 5: data = 8'b0010_0000; 6: data = 8'b0100_0000; 7: data = 8'b1000_0000; default: data = 'x; endcase	assign data = **1<<code;**

更に、また、従来では簡潔に表現できない条件でも、SystemVerilog の知識を活用すれば、簡潔明瞭、しかも汎用的な記述が可能になる場合があります。例えば、変数 v が 1、3、5、10、20、51 の場合に限りある機能を実行する処理の記述では、条件が複雑であるため適切な記述法に戸惑いがちです。しかし、SystemVerilog の知識があれば、このような処理は以下のように簡単に記述できます。そのためには、SystemVerilog の `inside` オペレータの知識が必要です。本書は、このような実践的な応用技術を養うための技術資料でもあります。

```
if( v inside {1,3,5,10,20,51} )
```

以上のような基本知識の活用法に加えて、検証分野で必要となる技法の解説も含まれています。例えば、IEEE Std 1800-2023 で追加された機能を適用してクラスを定義する事により、クラス記述の不注意な間違いを回避する技術が解説されています。また、virtual インターフェースを安全に、かつ簡単に操作するための技術の解説も含まれています。

この他、本書は組み合わせ回路とシーケンシャル回路のモデリング法と検証法を詳しく解説しています。モデリングでは、論理合成とのかかわりにおいて正しい記述法を導く解説をしています。検証法では、組み合わせ回路とシーケンシャル回路の出力を正しく取得するための適切なタイミングの知識を解説しています。

要約すると、本書は SystemVerilog の知識を実践に適用するための技術を総括した資料です。なお、少量ですが各章の終わりには練習問題が設けられているので、知識を確認する意味において問題を解いてみると良いです。第 10 章には全ての問題に対する解答が用意されているので、理解力を確認できます。本書を構成する各章を読むにつれて、新たな“ひらめき”が湧き出る読者も多いと思います。

アートグラフィックス
篠塚一也

変更履歴

日付	Revision	変更点
2024.09.08	1.0	初版。
2024.11.22	1.1	① virtual インターフェースを操作するためのクラスの記述を追加。 ② 汎用性を高める手段として static メソッドの使用法を追加。 ③ associative アレイの検索法を追加。 ④ 練習問題と解答を追加。

目次

1 概要

SystemVerilog は、設計分野および検証分野で使用されるハードウェア記述言語ですが、何れの分野で使用されるにしても、間違いのない正しい記述をしなければなりません。本章では、SystemVerilog の基礎知識だけでなく、技術者自身の創造力、独創性、“ひらめき”が適切な成果と新発見に結びつく例を紹介します。

設計作業では、ハードウェアをモデリングする目的で SystemVerilog を使用するため、使用できる命令や機能に制限があります。例えば、ハードウェアのモデリングに`#10` 等のディレーを指定できません。また、`fork/join`等のプロセスを生成する機能も使用できません。設計分野では、基本的には、コンパイル時に決定され得る情報だけでハードウェアのモデリングが行われなければなりません。これに対して、検証分野では SystemVerilog が備えている全ての機能を使用できます。したがって、検証分野では幅広い SystemVerilog の知識が要求されます。

設計分野では、機能的には SystemVerilog のサブセットが使用されるため、検証作業に比べると設計作業に求められる知識は、一見、単純になるような印象を与えますが、それは正しい結論ではありません。設計作業では、ハードウェアの動作を的確にモデリングする知識と技術が求められます。たとえ古くから踏襲されている記述法でも、現在の技術から判断をすれば、最適な手法とも言えない場合が多く見当たります。

1.1 論理合成とシミュレーション

SystemVerilog で設計したデザインは、回路に変換するために論理合成が使用されます。一方、設計したデザインの動作を検証するためには、シミュレーションが行われます。論理合成とシミュレーションは異なる目的で使用されますが、当然、処理方式も異なります。

先ず、論理合成は与えられたソースコードを実行する事ができないので、スタティックに解析します。一方、シミュレーションは、ソースコードを実行する事によりソースコードに書かれた動作を確認します。簡単な記述を使用して、論理合成とシミュレーションの相違を見てみます。以下に示す環境を仮定します。

```
logic [7:0]  a, b;
...
a = b;
```

シミュレーションは、代入文を合成する方法を持たないので、実行をして`b` の値を `a` にセットします。一方、論理合成は代入文を実行する事ができないので、スタティックな解析用の情報を作成する手段に使用します。つまり、`a` を `b` の代わりに使用できるようにします。言い換えれば、同値関係$a \equiv b$を確立して式の簡略化や最適化に使用します。物理的に表現すれば、`a` は`b` に配線で直接接続されるだけであり、回路は生成されません。

しかし、ビットセレクトやパートセレクトでは少し事情が異なります。以下の環境を仮定します。

```
logic [15:0] p;
logic [7:0]  z;
...
z = p[15:8];
```

ここで、上記の代入文を考察します。パートセレクト`p[15:8]`は`p` の一部なので、シミュレーションでは `z` に値を取り出す命令に変換する必要があります。しかし、論理合成では、以前と同様に同値関係を確立するだけです（表 1-1）。

表 1-1　SystemVerilog 記述に関するシミュレーションと論路合成の相違

シミュレーション	論路合成
`z = p>>8;`	$z \equiv p[15:8]$
シミュレーションでは、`p[15:8]`を抽出するためには、`p`の値を右に 8 ビットだけシフトして値を取り出さなければなりません。したがって、シミュレータ内部では、上記の命令に対応する処理が行われます。	論理合成では、`p[15:8]`の代わりに `z` を使用できるように情報を確立します。特別な回路は生成されません。しかし、論理の最適化が複雑になります。例えば、`z` が中間変数的な役割をしていると、`z[0]`が使用されていれば`p[8]`に置き換えられます。

このように、シミュレーションと論理合成は根本的に異なります。論理合成は、コンパイル時に決定できる情報のみを使用して回路に変換します。

1.2　設計作業に求められる知識と技術

先ず、古くから知られている記述法が、必ずしも適切な表現法ではない例を紹介する事から始めます。

例 1-1　慣習的な記述法の例

例えば、以下の Verilog 記述を考察してみます。変数 `tmp` のビットを右から左へ順に調べるために、シフトオペレータ（`tmp >> 1`）を使用しています。

```
function [3:0] count_ones;
input [7:0]  data_in;
reg [7:0]    tmp;
begin
   count_ones = 0;
   tmp = data_in;
   while( tmp ) begin
      count_ones = count_ones + tmp[0];
      tmp = tmp >> 1;
   end
end
endfunction
```

慣習化した記述法

古くから知られている手法なので、特別な違和感はないと思えますが、良く観察をしてみると、もう少し簡単な記述法でも同じ機能を実現できる事がわかります。例えば、以下のようにシフトオペレータを使用せずに書き換える事もできます。

```
tmp = tmp[7:1];
```

パートセレクト（`tmp[7:1]`）は、常に、符号なしなので、右辺は`{1'b0,tmp[7:1]}`と同じです。つまり、右辺は`(tmp>>1)`を表現しています。明示的に左辺と右辺のビット長を等しくするためには、以下のようにすれば良いです。

```
tmp = {1'b0,tmp[7:1]};
```

■

上記の例で指摘したかったのは、先入観にとらわれずにモデリングをしなければならないという点です。ビットをシフトする機能は、シフトオペレータ（<<、<<<、>>、>>>）だけでなく、パートセレクトによるビットスライスで実現できる場合が多く存在します。

先入観にとらわれない考え方に加えて、一度記述した内容を見直す習慣も大切です。その際、批判的な観点から見直す姿勢が大切です。次に、jk―フリップフロップの記述例を基にして、

2　データタイプ

本章では、デザインの実装や検証プログラムの実装に役立つデータタイプに関する基本知識をまとめます。基礎知識と云えば、データタイプ、および、その応用知識と言えます。データタイプに関する知識だけでなく、その使用され方によっては機能が異なる事を理解しておかなければなりません。

データタイプは、値の集合とそれらの値に適用されるオペレーションの集合です。数値としての値には、符号付きと符号なしがありますが、参照の仕方により変化します。例えば、ビットセレクトとパートセレクトは、常に、符号なしです。

2.1　整数系データタイプ

SystemVerilog には、整数系のデータタイプとして、logic、bit、byte、shortint、int、longint、integer、time[1]等がありますが、logic と bit 以外は、ビット数が予め定義されています。したがって、logic と bit 以外に対しては、ユーザがビット数を指定する事はできません。例えば、以下のような宣言はエラーになります・

```
int [31:0]        value;     // ILLEGAL
```

規定幅を持つ整数系のデータタイプに対して、基本的なデータタイプによる宣言法を表 2-1 にまとめておきます。

表 2-1　規定幅を持つ整数系のデータタイプと基本型による宣言法

規定幅を持つ整数系データタイプ	同等な宣言
byte	bit siged [7:0]
shortint	bit signed [15:0]
int	bit signed [31:0]
longint	bit signed [63:0]
integer	logic signed [31:0]
time	logic [63:0]

参考 2-1

packed次元の方向は意味を持ちます。例えば、byteは下記のBYTEにマッチしますが、ETYBにはマッチしません。

```
typedef bit signed [7:0] BYTE;
typedef bit signed [0:7] ETYB;
```

□

規定幅を持つ整数系データタイプは、packed 次元が宣言されているので。ビットセレクトやパートセレクトを使用できます。

例 2-1　int とビットセレクトとパートセレクト

変数 a が以下のように宣言されているとします。

```
int    a;
```

a は 32 ビットの符号付き整数です。符号なしの整数として扱うためには、a[31:0]とすれば

[1] SystemVerilog は、Verilog との互換性を保っているので、reg も使用できますが、SystemVerilog では、reg の使用を推奨していないので reg を除外しました。

良いです。また、LSB の 8 ビットを取り出すためには、a[7:0]を使用できます。符号を取り出すには、a[31]を参照すれば済みます。
■

2.2 数値と符号

logic は、それ自身では符号なしですが、signed を添えると符号付きになります。ネットや変数が符号付きとして定義されていても、参照の仕方により符号の状態が変わります。

例 2-2 データタイプの使用例

以下のような変数が宣言されていると仮定します。

```
logic [7:0]          a;
logic signed [7:0]   b;
integer              i;
time                 j;
```

変数の参照法により、機能が表 2-2 のように異なります。変数 b と i は符号付きですが、b[7:0]、{b}、i[31:0]、{i}は何れも符号なしになります。

表 2-2 参照の仕方による符号の変化

参照法	機能
a a[1] a[5:4] a[7:0] **b[7:0]** b[5:4] b[1] **{b}** **i[31:0]** i[7:0] **{i}** j j[63:0] j[7:0]	符号なし
b i	符号付き

■

では、何故、符号なし、または符号付きが重要な意味を持つ知識であるかを次に考えてみます。それには、古くから踏襲されている記述法 a>>>1 を説明するのが最もわかり易いと思います。

>>>オペレータは、左オペランドの符号を考慮します。つまり、左オペランドが符号付きで MSB が 1'b1 であれば、右にシフトする際に MSB を 1'b1 で埋めます。左オペランドが符号なしか MSB が 1'b0 であれば、右にシフトする際に MSB を 1'b0 で埋めます。したがって、左オペランドの符号は重要な意味を持ち、その特性を考慮して記述しなければなりません。表 2-3 は、シフトオペレータの機能を基本機能で表現し直した結果を示しています。

表 2-3　a >>> 1 と a >> 1 を基本機能で表現した例

宣言	a >>> 1	a >> 1
logic [7:0] a;	{**1'b0,**a[7:1]}	{**1'b0,**a[7:1]}
logic **signed** [7:0] a;	{**a[7],**a[7:1]}	

さて、シフトする機能は 1 ビットとは限りません。一般に、N ビットシフトする機能が必要になります。もし、N が定数であれば、上記に示した方法を表 2-4 のように一般化できます。

表 2-4　a >>> N と a >> N を基本機能で表現した例（N は定数）

宣言	a >>> N	a >> N
logic [7:0] a;	{**{N{1'b0}},**a[7:N]}	{**{N{1'b0}},**a[7:N]}
logic **signed** [7:0] a;	{**{N{a[7]}},**a[7:N]}	

ここで紹介した書き換え機能を応用するためには、{{}}オペレータに関する知識を持っていなければなりません。最後に、int 型の応用例を紹介しておきます・

例 2-3　int を packed アレイと見做した応用例

既に紹介したように int 型は以下のように bit の packed アレイです。

```
int    i; // same as bit signed [31:0] i;
```

変数 i に対して、表 2-5 のような記述法ができます。

表 2-5　int を bit の packed アレイと見做した応用例

標準的な記述法	packed アレイと見做した記述法
if(i < 0)	if(i[31])
i >>>= 2;	i = {{2{i[31]}},i[31:2]};
i >>= 2;	i = {{2{1'b0}},i[31:2]};

■

2.3　string データタイプ

string データタイプは設計で使用される事はありませんが、テストベンチや検証では重要な役割を果たすので復習しておきます。string は可変長の領域を保有するデータタイプなので、文字列を効率よく記憶する事ができます。簡単に言えば、string は byte 型の可変長アレイです。

2.3.1　string の機能

string データタイプは、8 ビット整数（byte）から構成される文字列です。C/C++の char*に相当しますが、const char*ではありません。C/C++と異なり、文字列の終わりにナル（\0）を含みません。また、文字列にナルを挿入しても無視されます。文字列の長さを N とすると、string 型の変数には、インデックスとして 0 から N-1 を使用できます。インデックス 0 は最初の文字を指し、インデックス N-1 は最後の文字を指します。初期値を設定しない string 型の変数は、長さ 0 の空文字列で初期化されます。例えば、以下のように宣言すると、name は空文字列で初期化され、title は文字列 "Test" で初期化されます。

```
string    name,
          title = "Test";
```

例 2-4　string 変数の定義例

変数を宣言するだけでなく、以下のように初期値の設定を行えます。

がって、通常のアレイコピー "B = A;" は以下の map 機能の簡略表現と言えます。

```
B = A.map(x) with (x);
```

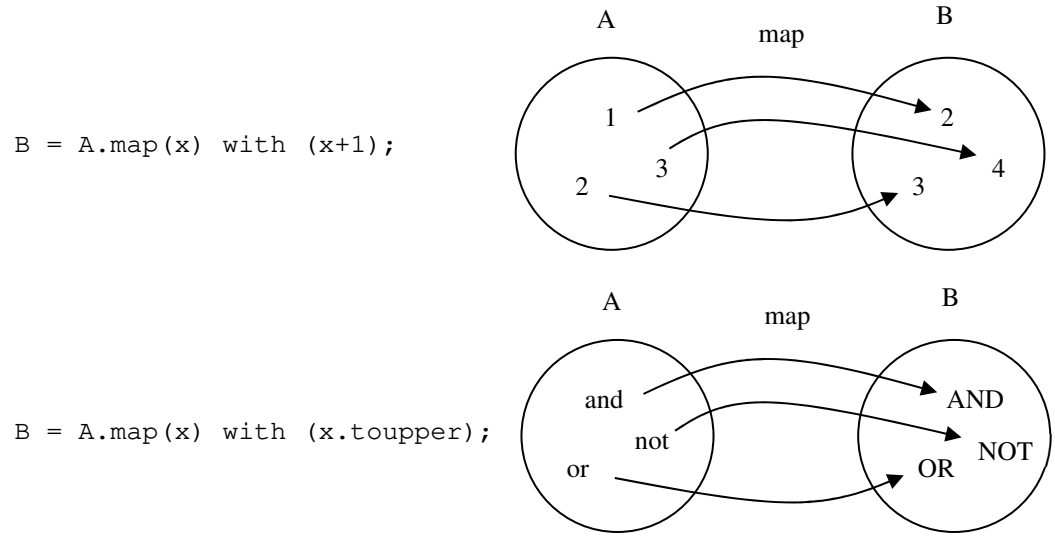


図 2-1　map 機能

以上の要点を以下の例にまとめておきます。

例 2-13　map()メソッドの使用例（[1]）

map()メソッドを以下のように使用して別のアレイを構築します。

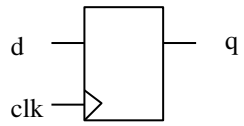
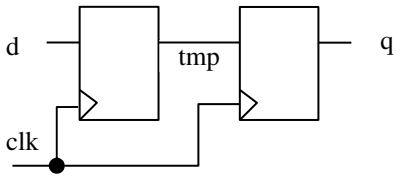
```
int A[] = {1,2,3}, B[] = {2,3,5}, C[$];

// Add one to each element of an array
A = A.map() with (item + 1'b1);          // A = {2,3,4}

// Add the elements of 2 arrays
C = A.map(a) with (a + B[a.index]);      // C = {4,6,9}

// Element by element comparison
bit Compare[];
Compare = A.map(a) with (a == B[a.index]); // Compare = {1,1,0}
```

■

2.16　unpacked アレイの操作メソッド

unpacked アレイのタイプによらず、便利なメソッドが定義されています。例えば、アレイ A の最大の要素を A.max()で求められます。処理条件を with で指定できるメソッドもありますが、条件指定を必須とするメソッドもあります。使用できるアレイメソッドを表 2-11 にまとめておきます。

表 4-1　ブロッキング代入文とノンブロッキング代入文の機能的相違

ブロッキング代入文を使用	ノンブロッキング代入文を使用
module design1(input clk,d, 　output logic q); logic　tmp; always @(posedge clk) begin 　**tmp = d;** 　q <= tmp; end endmodule	module design2(input clk,d, 　output logic q); logic　tmp; always @(posedge clk) begin 　**tmp <= d;** 　q <= tmp; end endmodule
tmp は一時的な変数として使用されているので、tmp にはフリップフロップが生成されません。	q は tmp を参照していますが、tmp には最新の d が設定されていないので、q は古い tmp を参照しています。したがって、tmp にもフリップフロップが必要になります。
	

ブロッキング代入文とノンブロッキング代入文を使用効果は、論理合成により正確に判定されます。上記の例が示すように、ノンブロッキング文の右辺は即座に左辺に有効になりません。

フリップフロップが生成されても、その出力が参照されていなければ最適化されて消滅してしまいます。

4.3　フリップフロップの最適化

簡単な例で論点を明確にします。以下の記述例を見て下さい。この記述により生成されるフリップフロップは q に対する 1 個のみです。

```
module design(input clk,d,output logic q);
logic  tmp;

always @(posedge clk)
    q <= d;

always @(posedge clk)
    tmp <= d;

endmodule
```

q はグローバル信号なので最適化されません

tmp は何処からも参照されていないので、最適化されて消滅します

tmp にもフリップフロップが生成されますが、tmp はモジュール外で参照されないため、論理合成により最適化されて消滅してしまいます。一方、q はモジュールの出力であるため、グローバル信号となり、モジュール外で参照される事になります。したがって、論理合成は q を最適化しません。

以上の論理は、以下の合成ルールを仮定している事に注意して下さい。

- 論理合成は、一般的に、モジュール毎に行われるので、合成中は他のモジュールからの参照を考慮する事ができません。したがって、論理合成はモジュールに記述された内容だけから参照関係を判断せざるを得ません。

7　モデリング

シーケンシャル回路にはフリップフロップが必要になりますが、それらの入力は組み合わせ回路で生成されるので、組み合わせ回路を如何に効率よくモデリングするかが問題となります。本章では、この問題を詳しく考察します。

7.1　パラメータによる汎用的記述

モジュールにパラメータを指定して定義すると、そのモジュールを使用する側でパラメータの値を変更できます。したがって、一つのモジュール定義から異なる構造を持つモジュールのインスタンスを作る事ができます。

例 7-1　パラメータによる汎用的な記述例

以下の記述にあるように、パラメータの指定をモジュール宣言の先頭で行います。そして、定義されたパラメータを使用してポートの宣言をします。以下の記述は 4 ビットの加算器ですが、NBITS に 16 を割り当てれば 16 ビットの加算器を表現する事になります。

モジュールの先頭にパラメータを定義する

```
module adder #(NBITS=4)
    (input logic [NBITS-1:0] a,b,output co,[NBITS-1:0] sum);
assign {co,sum} = a+b;
endmodule
```

例えば、以下のようにすると adder を 16 ビットの加算器として使用できます。

```
adder #(.NBITS(16)) ADDER(.*);
```

■

7.2　記述したコードの吟味

一旦記述しても見直す作業は必要です。ここでは、簡単な回路を使用して、コードを見直す効果が大きな改善に結び付く事を解説します。ここで示す例は、以下のようなアップダウンカウンターの例です。

```
module ud_counter(input clk,reset,[1:0] up_down,
    output logic [3:0] count);

always @(posedge clk,posedge reset)
    if( reset )
        count <= '0;
    else if( up_down == 2'b00 || up_down == 2'b11 )
        count <= count;
    else if( up_down == 2'b01 )
        count <= count + 1;
    else if( up_down == 2'b10 )
        count <= count - 1;

endmodule
```

up_down が 1 であればカウントアップ、2 であればカウントダウン、up_down が 0 または 3 であれば現在の値を保持するシーケンシャル回路です。多少の無駄な記述はあるものの論理合成が最適化してくれるので特に気にならないと思います。

しかし、矢張り無駄な記述が多いので見直す必要がありそうです。例えば、up_down が 2'b00 または 2'b11 という事象は、up_down[1]==up_down[0]で表現できます。同様に、up_down==2'b01 は up_down[1]<up_down[0]と書けます。したがって、上記の always ブロックを以下のように書き換えられます。

```
always @(posedge clk,posedge reset)
    if( reset )
        count <= '0;
    else if( up_down[1] == up_down[0] )
        count <= count;
    else if( up_down[1] < up_down[0] )
        count <= count + 1;
    else if( up_down[1] > up_down[0] )
        count <= count - 1;
```

さて、これで完成ではありません。この書き換えでは、==、<、>の 3 つのオペレータが同時に使用されていますが、明らかに無駄です。何れかの 2 つのオペレータを使用するだけで十分です。したがって、以下のように書き換えができます。

```
module ud_counter(input clk,reset,[1:0] up_down,
    output logic [3:0] count);

always @(posedge clk,posedge reset)
    if( reset )
        count <= '0;
    else if( up_down[1] < up_down[0] )
        count <= count + 1;
    else if( up_down[1] > up_down[0] )
        count <= count - 1;

endmodule
```

このように、一度書いたコードでも見直すと改善に結ぶ着く事があります。

7.3 発想転換の重要性

1.1 節で述べたように、古くから知られている方法が最適なモデリング法とは限りません。時代に即した手法を採用する必要があります。デコーダーに関しては、古い昔から RTL 記述はアレイを展開した形式で行うのが一般的です。これは、主として、論理合成を意識しているためですが、一般のプログラミング技術に相反する習慣です。

例えば、3:8 デコーダーを考えてみます（表 7-1）。decoder1 の記述には違和感が無いと思います。しかし、C/C++ではこのような記述法を決してしません。寧ろ、decoder2 の方が C/C++のプログラミング記述法に近くなります。decoder1 はビット数に関してハードコーディングされているので、ビット数に変更があれば、全面的な書き換えが必要になります。一方、decoder2 は汎用的な手法を用いています。

表 7-1　3:8 デコーダーに記述法

標準的なデコーダー記述	一般のプログラミング技術による記述
module decoder1(input logic [2:0] code, output logic [7:0] data); always @(code) case (code) 0: data = 8'b0000_0001; 1: data = 8'b0000_0010;	module decoder2#(NBITS=8)(input logic [$clog2(NBITS)-1:0] code, output [NBITS-1:0] data); assign data = 1<<code; endmodule

8　検証

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。

8.1　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 8-1　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 8-1）。

表 8-1　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。

8.2 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)[2] 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 8-1）。

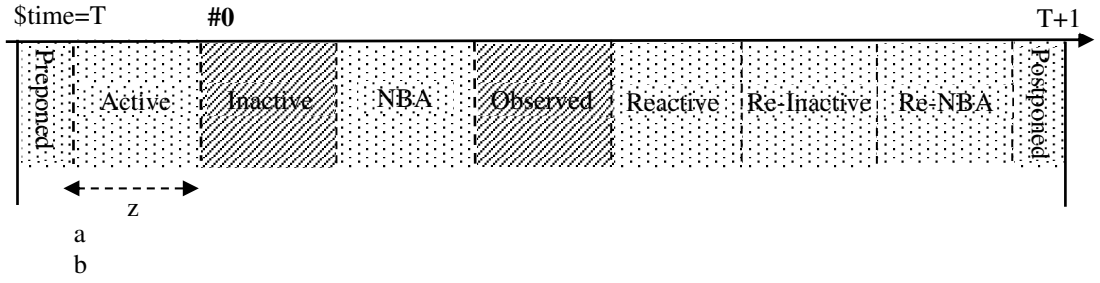


図 8-1 組み合わせ回路の出力と Active 領域

次に、具体的に時間差が発生する状況を例示し、組み合わせ回路の出力を検証する最適なシミュレーション領域を考察します。

例 8-2 組み合わせ回路の出力は直ぐには安定しない例

どのような組み合わせ回路に対しても共通する現象なので、以下の簡単な 1 ビットハーフアダーを利用して説明します。

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